

ARCHITECTING SOCIAL INTERNET OF THINGS

by

Ji Eun Kim

B.E. Dong-A University, South Korea, 1995

M.S. Carnegie Mellon University, USA, 2003

Submitted to the Graduate Faculty of
the Dietrich School of Arts and Sciences in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2015

UNIVERSITY OF PITTSBURGH
DIETRICH SCHOOL OF ARTS AND SCIENCE

This dissertation was presented

by

Ji Eun Kim

It was defended on

November 23rd, 2015

and approved by

Daniel Mosse, Ph.D., Computer Science Department, Professor

Shi-Kuo Chang, Ph.D., Computer Science Department, Professor

Jingtao Wang, Ph.D., Computer Science Department, Assistant Professor

Rosta Farzan, Ph.D., School of Information Sciences, Assistant Professor

Dissertation Director: Daniel Mosse, Ph.D., Computer Science Department, Professor

ARCHITECTING SOCIAL INTERNET OF THINGS

Ji Eun Kim, PhD

University of Pittsburgh, 2015

In the new era of the Internet of Things (IoT), most of the devices we interact with daily are connected to the Internet. From tiny sensors, lamps, home appliances, home security systems and health-care devices, to complex heating, ventilation and air conditioning (HVAC) systems at home, myriad devices have network connectivity and provide smart applications. The *Social Internet of Things* (SIoT) is a new paradigm where IoT merges with social networks, allowing people and connected devices as well as the devices themselves to interact within a social network framework to support a new social navigation. Smart homes is one of the domains that can fully leverage this new paradigm, which will enable people and devices, even in different homes, to actively and mostly automatically collaborate to discover and share new information and services. Unfortunately the heterogeneous nature of the devices around the home prohibits seamless communication in the (S)IoT. Furthermore, the state-of-the-art solutions in smart homes offer little, if any, support for collaborating users and devices. This dissertation describes a new, scalable approach to connect, interact and share useful information through devices and users with common interests. The dissertation has three contributions. First, it proposes a holistic and extensible smart home gateway architecture that seamlessly integrates heterogeneous protocol- and vendor- specific devices and services and provides fine-grained access controls. Second, it defines an interoperable, scalable and extensible software architecture for a novel cloud-based collaboration framework for a large number of devices and users in many different smart homes. Third, it provides a reasoning framework to enable automated decisions based on the discovered information and knowledge created and shared by end users. The developed architecture and solutions

are implemented in real systems, which integrate with many different devices from different manufacturers and run multiple categories of rules created by end users. The architectural evaluation results show the developed systems are interoperable, scalable and extensible.

TABLE OF CONTENTS

PREFACE	xii
1.0 INTRODUCTION	1
1.1 PROBLEM DEFINITION	2
1.2 CONTRIBUTIONS	3
2.0 RELATED WORK	5
2.1 SMART HOME ARCHITECTURES	5
2.2 SEMANTIC MODELS FOR SMART HOMES	6
2.3 SMART HOME ACCESS CONTROL	8
2.4 SOCIAL INTERNET OF THINGS	9
2.5 REASONING ENGINES	10
2.6 END USER PROGRAMMING FOR SMART HOMES	11
3.0 INTEGRIFY: SEAMLESS INTEGRATION OF HETEROGENEOUS DEVICES IN SMART HOMES	13
3.1 REQUIREMENTS FOR SMART HOME ARCHITECTURE	14
3.2 SMART HOME ARCHITECTURE	16
3.2.1 Smart Home Gateway Architecture	17
3.2.1.1 Background of OSGi	18
3.2.1.2 Home Network Communication Technologies	18
3.2.1.3 Smart Home Device Stack and Discovery	20
3.2.2 Message Framework	23
3.2.3 Semantic Integration	25
3.2.3.1 Reasoning Engines	26

3.3	ACCESS CONTROL FOR SMART HOMES	28
3.3.1	Policy Model	28
3.3.2	Access Control Architecture and Design	29
3.4	INTEGRIFY IMPLEMENTATION: PROOF OF CONCEPT	30
3.4.1	Implementation	32
3.4.2	Evaluation	34
4.0	SOCIALITE: A CLOUD BASED DISTRIBUTED COLLABORATION FRAMEWORK FOR SOCIAL INTERNET OF THINGS	37
4.1	NEW SOCIAL RELATIONSHIPS AND APPLICATIONS	38
4.1.1	New Social Relationships	38
4.1.2	New Applications Leveraging New Social Relationships	40
4.2	USER SURVEY: SOCIAL INTERNET OF THINGS FOR SMART HOME SYSTEMS	42
4.2.1	Methodology	42
4.2.2	Demographics	44
4.2.3	Categorization of SIoT Features	46
4.2.4	Observation of Relationship Types and Device Life-cycle Relevance . .	49
4.2.5	Acceptance of SIoT, End User Programming and Sharing Rules	49
4.3	NON-FUNCTIONAL REQUIREMENTS	51
4.4	SOCIALITE SYSTEM OVERVIEW	52
4.5	SOCIALITE SEMANTIC MODELS	54
4.5.1	Background of Technologies in Semantic Web	54
4.5.2	Socialite Semantic Model Description	55
4.6	SOCIALITE SERVER ARCHITECTURE	58
4.6.1	Interoperability for Various Manufacturer's APIs	59
4.6.1.1	Physical and Logical Devices	59
4.6.1.2	Common Device Model and Manufacturer Specific Device Im- plementation	61
4.6.2	Persistent Management and Repositories	61
4.6.2.1	Semantic Model Management and Its Repository	61

4.6.2.2	Device History Management and Its Repository	63
4.6.2.3	Rule Management and Its Repository	64
4.6.3	Scalability with Event-Driven Architectural Pattern	65
4.6.3.1	Background of Event-Driven Architectures	65
4.6.3.2	Socialite Event Channels	66
4.6.3.3	Socialite Event Generators	67
4.6.3.4	Event Processing Styles	70
4.6.4	Large Scale Reasoning over Data Streams	71
4.6.4.1	Background of Apache Storm	72
4.6.4.2	Socialite Storm Topology	73
4.7	END USER EMPOWERED REASONING FRAMEWORK	75
4.7.1	Socialite Reasoning Framework	77
4.7.1.1	Background of Reasoning Engines	77
4.7.1.2	Socialite Reasoning Concept	80
4.7.1.3	Rule Transformation to Domain Specific Language	88
4.7.1.4	Rule Management and Sharing	89
4.7.2	Socialite Client Application	92
4.7.2.1	Features	92
4.7.2.2	End User Programming for Rules	94
4.7.2.3	Architecture Overview	96
4.8	SOCIALITE IMPLEMENTATION: PROOF OF CONCEPT	96
4.8.1	Implementation	96
4.8.2	Evaluation of Architecture	101
4.8.2.1	Interoperability Evaluation	101
4.8.2.2	Scalability Evaluation	102
4.8.2.3	Extensibility Evaluation	104
5.0	CONCLUSIONS AND FUTURE WORK	106
	BIBLIOGRAPHY	109

LIST OF TABLES

1	Statistics for execution time (ms) of service calls to lamp device	35
2	New proposed relationship types in Socialite	39
3	Fields grouping input distribution per user	103

LIST OF FIGURES

1	High level architecture for smart home systems	16
2	Core software building blocks for the home gateway	19
3	Device stack (a, left) and an instance of Insteon device (b, right)	21
4	New controller discovery process for the unknown controller driver to the gateway	22
5	ZigBee end device discovery process	24
6	<i>Integrify</i> message framework	25
7	Excerpt of the ontology representation	27
8	Access control design concept	30
9	<i>Integrify</i> demonstrator	32
10	Execution time of service calls for the different experimental settings	35
11	Graphical representation of the new social relationships in <i>Socialite</i>	39
12	Demographics of participants	45
13	Distribution (%) of the nine feature categories from the user survey analysis .	46
14	Distribution of programming experience, end user programming acceptance, sharing rules, SIoT acceptance	50
15	<i>Socialite</i> system overview	52
16	RDF graph with two nodes (Subject and Object) and a triple connecting them (Predicate)	55
17	Graphical representation of the core ontologies in <i>Socialite</i> (Shortend)	56
18	Example of physical and logical devices of thermostat	60
19	Common semantic models for devices	60
20	Broker topology in event-driven architecture	66

21	Status update from (1) the physical devices, (2) the client application, and (3) from the reasoning engine	69
22	Storm cluster concept	73
23	Storm topology in <i>Socialite</i>	74
24	Event-driven distributed architecture	76
25	High level view of production rule system [18]	78
26	Mapping categorized features from the user survey to rule categories and other functions	81
27	Intermediate common data models between the data payload in REST APIs and Drools domain specific language	89
28	Rule sharing concept	91
29	<i>Socialite</i> Web based user interfaces	93
30	End-user programming user interface	95
31	<i>Socialite</i> client architecture overview with a rule creation example	97
32	Dynamically accessible <i>Socialite</i> APIs	99
33	<i>Socialite</i> deployment view on Amazon Web Services (AWS)	100

LIST OF CODES

1	Example of XACML policy (shortened): Permit kids to turn on entertainment devices only before 7 PM	31
2	Example of Java class with JenaBean annotation	62
3	Example of resource representation in Turtle format	63
4	Basic structure of the rule model in the production rule system	79
5	Context generation rule example	83
6	Automation example with user-defined context	84
7	Example of JSON payload	90

PREFACE

I would like to express my special appreciation and thanks to my advisor, Professor Daniel Mosse, you have been a tremendous mentor for me. I would like to thank you for encouraging and guiding my research for a long journey of my Ph.D. life. You trust me and share your life experience even when I had difficulties in balancing out my Ph.D. research and professional work. You are always available when I needed your advice and support.

I would also like to thank my committee members, Professor Shi-Kuo Chang, Professor Jingtao Wang, and Professor Rosta Farzan for serving as my committee members. My dissertation topics include various research areas where your advice on software engineering, human computer interaction and social computing are great help to proceed my research and shaped my dissertation a lot.

I would like to thank my current company Bosch and colleagues in our research center. Bosch had financially supported my Ph.D. study and research. Discussions with my colleagues always inspired me to develop a better idea than the ones from yesterday.

I also realize that Ph.D. is one of socialable works in my life. I would like to thank the many students and researchers who had worked with me in discussions, software development and writing papers together.

I would like to thank my family far away in South Korea, who is the most important reason for me to get through long Ph.D. life with great joy. Your prayer is a deep foundation for who I am.

Finally, I thank my God, my good Father, for letting me through all the difficulties. I had experienced Your guidance day by day. You are the one who let me finish my degree.

I will keep on trusting You for my future. Thank you, Lord.

1.0 INTRODUCTION

We are moving toward the new era of the Internet of Things (IoT) [160] where most of the devices we constantly interact with have an Internet connection. Devices from tiny sensors, lamps, home appliances, home security systems and health-care devices to complex Heating, Ventilating and Air Conditioning (HVAC) systems at home have network connectivity. These devices are remotely accessible for people to monitor and control, and to run smart applications. Gartner estimates that there will be nearly 26 billion devices on the IoT by 2020 [122]. According to a study released by Juniper Research, the smart home market, equipped with these devices will reach \$71 billion by 2018 [157].

A parallel and also explosive trend is that of social networks: most people are already connected through various social networking services. With the help of the growing ubiquity of smart phones, people are constantly generating content and easily sharing it with others in their social networks. According to Pew Research Center, 74% of online adults use social networking services as of January 2014 [6]. Another surprising fact is that, as of January 2015, more than half (56%) of all online adults 65 and older use Facebook [113].

The *Social* Internet of Things (SIoT) [130, 30, 133] merges the IoT with social networks. It is a new paradigm that allows people and connected devices, as well as devices themselves, to interact within a social network framework that supports a new practice in social navigation [71, 69, 126]. It is expected that the SIoT will facilitate the seamless connection and unprecedented collaboration of people and devices through new social relationships, which has not yet been foreseen. The devices in the SIoT will become active participants based on their new *social* roles in various contexts. The information generated and processed from and directed to devices/people will become sharable with connected participants of the SIoT. Furthermore, people and devices will not only become more cooperative but will also gain the

ability to make automated decisions with their knowledge in a distributed and cooperative manner.

Smart homes is one of the domains that can fully leverage this new paradigm. This dissertation propose a new collaboration framework for SIoT focusing on smart homes.

1.1 PROBLEM DEFINITION

A smart home user study [53] uncovered that there are four barriers for broad adoption of smart homes, namely high cost of ownership, inflexibility, poor manageability and difficulty in achieving security. In addition, I note that the heterogeneous nature of the smart home system as well as the diverse groups of users are intrinsic causes of these barriers. In fact, the lack of a *de facto* communication standard for smart home devices creates these barriers, hindering the interoperability of devices from different vendors. In addition, more diverse types of devices connected to the home result in more diverse groups of users that interact with smart home solutions. For example, health-care related devices can be remotely accessible by health-care service providers. Given the various groups accessing these devices, the development of a flexible yet fine-grained access control mechanism is required to securely manage various smart homes with different configurations of devices for diverse groups of users.

It is expected that devices from different manufacturers will interact with each other transparently in the Social Internet of Things. However, each manufacturer provides different application programming interfaces and data models for their connected devices. Therefore, one of the big challenges is how to utilize all the functionality coming from different devices with similar capabilities and to provide a proper abstraction for implementation details of accessing different programming interfaces and data models.

Even if devices from different manufacturers were able to communicate seamlessly with each other, no mechanisms currently exist to effectively discover and share useful and relevant information at scale. It is possible that neither an owner of a device, nor a device itself knows if they could choose a more appropriate configuration for the device's parameters, for example

to improve energy efficiency. However, if all the devices with same capabilities were able to share their information with other devices in their social networks, an individual device would have the opportunity to learn the configuration that results in a better operation.

Last but not least, there is no established approach for how to use the discovered information from the participants, both devices and humans to achieve common goals in a collaborative way. The state-of-the-art solutions are inflexible and do not leverage the device functionalities/capabilities for new cross-domain applications or services. For example, currently an alarm can only be detected only by dedicated devices (e.g., motion detection sensor) installed for the home security system, instead of exploiting the same capabilities available in other devices (e.g., motion detection sensor embedded in a thermostat). Furthermore, an issued alarm is dispatched to only a specific security service provider rather than flexibly dispatched to a neighbor or to a nearby police officer. As shown in this example, only specific users and devices are used as designed by the manufacturer with less flexibility. Therefore, current solutions limit the potential participation of users and devices for a collaborative situation.

1.2 CONTRIBUTIONS

This dissertation creates a new approach and proposes a new solution to connect, interact and share useful information through devices and users having common interests at scale. The developed system aims to provide an interoperable, scalable and extensible collaboration framework where smart home devices in a new social framework are able to make automated decisions over the discovered information in a distributed manner, based on rules that are created and shared by end users. To the best of the author’s knowledge, the developed solution in this dissertation is the first smart home system enhanced with the Social Internet of Things (SIoT) and integrated with heterogeneous real devices, which enables unprecedented people-device and device-device collaborations, based on the end user defined and shared rules.

In particular, the contributions of this dissertation are as follows.

- A holistic and extensible **smart home gateway architecture** that allows heterogeneous devices to be flexibly and dynamically installed, managed and accessed during runtime
- A new **access control mechanism** specific to smart home systems that utilize the user's role, the device status, the device location and the current time as access control attributes
- An **interoperable, scalable and extensible software architecture** for a new cloud-based collaboration framework that realizes the new features for the Social Internet of Things through integration of real smart home devices from various manufacturers
- **Semantic models** for users, devices and their diagnostics, locations, services, and new social relationships to be the basis for the interoperability of the devices as well as autonomous interactions between people and devices, and between devices themselves
- A new **reasoning framework** that uses the semantic models for the basic/low-level knowledge representation, events/messages for asynchronous communication, and production rules for high-level reasoning. The reasoning capabilities include device and capability-based automation, context-based automation, preference-based automation, context generation and service invocation. New social relationships and temporal aspects are expressible in these reasoning capabilities.
- A new **end-user programming tool** that enables end users to create and share new rules: these rules are fully integrated with the reasoning framework

The rest of this thesis is organized as follows. Chapter 2 summarizes related work. Chapter 3 and Chapter 4 describe proposed approaches from this dissertation and present the proof of concept. Finally, the conclusion of this work and the future research directions are put forth in Chapter 5.

2.0 RELATED WORK

Related work is categorized into the five research areas: 1) smart home architectures, 2) semantic models for sensors, devices and services, 3) smart home access control mechanisms, 4) Social Internet of Things paradigm and related research and 5) various reasoning approaches. In the following sections each of these areas are discussed separately.

Note that the background knowledge employed to the system as basic foundation such as communication protocols, principles of technologies, commonly used open-source or commercialized products are discussed separately in relevant chapters.

2.1 SMART HOME ARCHITECTURES

Despite the heterogeneous nature of home network standards, existing smart home research generally assumes a homogeneous underlying architecture. MavHome [60], for example, predicts activities in a home and makes the home act as an intelligent agent providing optimal support for its inhabitants. In [90], the authors address the important role of context for smart home applications by providing adaptive middleware and APIs that provide context to applications. Projects in [94] and [170] aim to assist end users to build their own individual smart home applications. While these visions are important for the success of smart homes, dealing with the heterogeneity of devices and services is a crucial requirement for them to be realized.

The Open Service Gateway initiative (OSGi) [22] has been used for various home automation solutions [70, 168, 164, 87, 82, 107] as a basic framework. There are a number of similar architectural approaches to our smart home gateway architecture discussed in

Chapter 3, which targets interoperability on the network protocol level. For instance, DOG (Domotic OSGi Gateway) [45], Hydra [73], Amigo [79] and a project in [136] are the most relevant approaches to our smart home gateway architecture in Chapter 3. All of them are based on OSGi [22] and use a semantic model for abstraction. The device interoperability is achieved by a multi-layer device stack, which roughly consists of drivers, common adapters for similar device types and a high-level representation described semantically by a domain model. Unfortunately, none of them gives a clear account on how devices communicating over different protocols are discovered.

To address these issues, we present a complete device discovery workflow in smart home: plug and play of heterogeneous devices during runtime, extensibility to new devices that are not foreseen during system development, and the use of a barcode reading functionality with smart phone cameras as a unique approach to improve usability. Moreover, our approach in Chapter 3 integrates cloud services to increase coverage of the device discovery process and to extend the smart home functionality with new applications, drivers, and computationally intensive services. The authors of [167] present an architecture to integrate cloud services and smart home networks. However, unlike the smart home gateway in Chapter 3, they did not implement the architecture in a prototype.

2.2 SEMANTIC MODELS FOR SMART HOMES

The DogOnt [45] ontology supports device and network independent descriptions of houses, including both controllable (e.g., home appliances) and architecture (e.g., floor plan) elements. It is designed with a particular focus on inter-operation between domotic systems by modeling device, state, functionality and network. The DogOnt ontology is built by inheriting the concepts available in DomoML [156], which is a mark-up language aiming at the definition of an interoperability standard for domestic resources. The DogOnt ontology is used in our smart home gateway discussed in Chapter 3 by extending it with more functionality concepts that are needed for our system.

The Hydra [73, 150] project extends the FIPA device ontology [76] and vocabularies

from the Amigo project for device descriptions [79]. The Hydra device ontology includes the basic device information, device services, device events, device malfunctions and device capabilities such as hardware properties and state machine.

Standard Ontology for Ubiquitous and Pervasive Applications (SOUPA) [55] is an ontology that includes modular component vocabularies to represent intelligent agents with associated beliefs, desires, intentions, time, space, events, user profiles, actions, and policies for security and privacy. Although they present applications and extensibility used in smart spaces, SOUPA ontology addresses generic models rather than domain specific concept descriptions.

Semantic Web technologies have been proposed as a means to enable interoperability for sensors and sensing systems [59]. The Sensor Web Enablement (SWE) initiative of the Open Geospatial Consortium (OGC) [47] defined data encoding and Web services to store and access sensor-related data. These standards such as SensorML [48] and Observation and Measurement [132] provide syntactic interoperability [153]. The World Wide Web Consortium (W3C) Semantic Sensor Network group defines an OWL2¹ [80] ontology to describe the capabilities and properties of sensors, the act of sensing and the resulting observation [59]. Unfortunately these ontology models are limited to sensors. Therefore, none of these models completely supports all required semantic models for the smart home systems consisting of sensors, actuators and devices with multiple capabilities.

Semantic Web technologies have been used also for describing Web Services. OWL-S [117] is an ontology built on top of OWL for describing Semantic Web Services to automatically discover, invoke, compose and monitor Web resources offering services, under specified constraints. OWL-S is used for the Web Service Description Language (WSDL) [58], which is an XML format for describing network services as a set of endpoints. It does not aim for the current trend of REST Web Services [144], which is a lightweight implementation of service oriented architecture. Similarly Web Service Modeling Ontology (WSMO) [146] and Semantic Annotation for WSDL [106], officially supported by W3C describe only WSDL based Web Services. The hRESTS (HTML for RESTful Services) [105] is a microformat for machine-readable descriptions of Web APIs. The hRESTS microformat describes main as-

¹OWL stands for Web Ontology Language

pects of services, such as operations, inputs and outputs. *Socialite*, a cloud based framework for Social Internet of Things discussed in Chapter 4 reuses the vocabularies used in hRESTS described in a microformat but converts them to an RDF model to be compatible with other semantic models in the *Socialite* system.

2.3 SMART HOME ACCESS CONTROL

The Role Based Access Control (RBAC) concept began with multi-user and multi-application online systems pioneered in the 1970s [149]. The authors in [101] articulate that the fundamental challenge is how to enable home users to manage access control policies for everyone who visits their homes. The main issues revolve around the complexity and diversity of the resources, the diversity of the subjects, the low sophistication of the administrators, and the social context.

Several studies suggested the access control mechanism for the home network environment in which the OSGi service platform is operated. The authors in [57] proposed an authorization policy management framework based on RBAC for OSGi service platform. They classify the policy into two types: user-role assignment policy and permission-role assignment. They claim that RBAC model is flexible and more proper than other models such as discretionary access control (the owner of an object has discretionary authority over the others) [148] for home network environments operated by the OSGi service platform. The authors in [20] provide RBAC for the OSGi service environment and use eXtensible Access Control Markup Language (XACML) [139, 124] for their policy descriptions.

However, the authors do not address other important attributes in our policy model for the smart home gateway, such as device type, location, device status and current time. Our policy model provides a fine-grained access control with these attributes.

2.4 SOCIAL INTERNET OF THINGS

A few researchers recently have investigated the convergence of the Internet of Things (IoT) and social networks as a promising direction for the future IoT systems [130, 30, 27, 29]. The new paradigm of the Social Internet of Things (SIoT) supports novel applications and networking services for the IoT in a more effective and efficient way [30]. The authors in [133] summarize the literature review on the ongoing activities in the SIoT and address open research challenges in the generic software architecture.

Earlier approaches in [33, 85] employed human social networking sites as a medium for publishing and sharing the measured data. However, the devices are not considered as actors who can directly interact to share the information.

In [114], social devices for co-located devices and humans were introduced to interact with each other. Their approaches aimed to enhance the remote communication for the social media services when users and devices are located in the same space. Similar to our social relationship models, the work in [29] devised different relationships between human and devices. It discusses the high-level conceptual architecture model including the core components of the system (e.g., ID management, service discovery and composition, and trustworthiness management). Unlike our solution, their architecture addresses only the security aspect of the system, which is a complement for our solution, and they did not consider other important non-functional requirements such as interoperability, scalability and extensibility.

Paraimpu [137] is a platform for a Social Web of Things that allows sharing of the objects with other users. It allows the mash-ups of different devices by composing REST Web Services. Similarly, the SenseWeb project [93] develops a platform for people to share their sensor readings using Web services to send the data to a central server. However, none of these works addresses the heterogeneity of devices and APIs from different manufacturers, and use a common semantic model to mitigate this problem. Furthermore, their approaches do not include different social relationships and thus the possible collaboration of users and devices is limited.

2.5 REASONING ENGINES

Semantic models have been adopted by various researchers to enable smart applications such as smart homes and buildings [99] and energy-efficient homes [142]. The ontology model and inference rules are used in the middleware to infer the contextual information of the user of the system [83, 166, 165].

Semantic reasoners are practically available as software frameworks and tools [44]. Pellet [154] reasoner is a Description Logic (DL) reasoner that implements tableau based algorithms [31] for the inference calculus. Sesame [52] and OWLIM [102] use standard rule engine to reason with OWL.

Our smart home gateway system discussed in Chapter 3 integrates these semantic reasoners to discover knowledge, which is not explicitly specified in the smart home gateway ontology but can be inferred based on the description logic from OWL data model (e.g., transitive and symmetric properties).

Our cloud based collaboration framework discussed in Chapter 4 approach requires a mechanism to reason not only based on current facts, but also based on the history of the previous events for temporal reasoning. The related work is covered in both semantic model based reasoning as well as other reasoning engines without semantic models such as production rule systems and Complex Event Processing (CEP).

SQL-like and algebra-based event languages are designed to specify the semantics of events [130], but they also lack solid support from event processing engines. The authors in [145] present Time-Annotated Resource Description Framework (TA-RDF) as a formal extension to the RDF data model, which attaches a timestamp to each group of RDF triples. The authors of [25] extend the standard SPARQL [140] query language by adding binary temporal operators, so that semantic CEP can be done by executing Event Processing SPARQL (EP-SPARQL) queries. Obviously, both solutions [145] [25] require modifications and optimizations of SPARQL query engines and RDF repositories.

The Rete algorithm [77] is a pattern matching algorithm used in production rule systems such as Drools[18] of which our *Socialite* system builds the reasoner on top. CEP [110, 169] combines data from multiple sources to infer meaningful events or patterns, and responds to

them as quickly as possible.

The authors in [109] propose a framework where Linked Data are first translated into events conforming to a lightweight ontology, and then fed to CEP engines to bridge the gap between Semantic Web technologies and CEP. Their approach is similar to the *Socialite* reasoning framework in a sense that they use both Semantic Web technologies and CEP. However, the *Socialite* reasoning framework is different because we use the semantic model for basic/low-level knowledge representation (e.g., device and people), events/messages for asynchronous communication between event sources and reasoning engines, and production rules for the high-level reasoning including temporal reasoning with CEP.

Since the low-level knowledge is based on the ontology model, our reasoning discussed in Chapter 4 supports not only a device specific automation, but also a capability based automation. Furthermore, none of the previous work considers using production rules with social relationships and device capabilities that facilitates collaboration to efficiently share configuration and information even with others' devices otherwise unknown to the user.

2.6 END USER PROGRAMMING FOR SMART HOMES

A number of researchers have investigated end user programming in smart homes [163]. The authors in [65] analyzed 47 papers from ACM and IEEE Xplore Digital Libraries for end user programming, and proposed the set of guidelines recommending trigger-action or trigger-constraint-action formats for representing smart home rules. Furthermore, the real-world applications adopting the end user programming with connected devices and Web services have become popular over the last years. IFTTT [12], Atooma [9] and WigWag [17] are examples that are used in industry.

The authors in [68] interviewed 20 participants about a context-aware application called iCAP, finding the most common mental models to be rule-based as in trigger-action programming (“if something happens, then do something else”). OSCAR [129] is an application that supports flexible and generic control of devices and services in near-future home media networks that also employs trigger-action programming. The authors in [163] confirmed

that many desired behaviors can be expressed using trigger-action programming through their user study with IFTTT community usage.

In contrast, the authors in [53] argue that trigger-action programming can be difficult for users to debug when problematic behaviors inevitably occur. The authors in [141] suggest using machine learning over end-user programming.

The authors in [63, 64] studies users (family) in smart home rather than a single user and articulate that the home is collaborative and conflict resolution is complex for the group of users in home.

Although these studies address end user programming either for a single user or family, no research has been done for the end user programming for smart homes with consideration of social relationships. Our user survey and the end user programming prototype in Chapter 4 are the first research contribution that incorporate various categories of rules for Social Internet of Things in smart home applications to the best of our knowledge.

3.0 INTEGRIFY: SEAMLESS INTEGRATION OF HETEROGENEOUS DEVICES IN SMART HOMES

Smart home systems provide automation capabilities that allow home owners to have more complete control over their home, and promote energy efficiency that allows them to save money on energy bills. These smart home solutions integrate many devices with different capabilities such as intrusion detection, video surveillance, fire detection, patient health monitoring and entertainment. Many of these devices use different communication protocols that are mostly incompatible with each other. Examples of such protocols are power line communications such as X10 [147] and Insteon [62], wireless communications (ZigBee [24], Z-Wave [23]), IP-based UPnP (Universal Plug-and-Play) [138], SOAP-based web Services on devices such as DPWS (Device Profiles for Web Services) [4], Web of Things [86] using RESTful Web services and many more proprietary protocols from diverse manufacturers.

Although the market prediction and current technological trends look promising, we observed that no *de facto* communication standard exists in the smart home. This hinders the integration of different services (e.g., energy management, security). Therefore, we propose a smart home software architecture based on the OSGi (Open Services Gateway initiative) framework [22], which seamlessly integrates heterogeneous protocols and diverse device types used in home networks. Our aim is to enable end users to add new devices on demand, regardless of the discovery peculiarities imposed by the particular communication protocol. In addition, since device access by applications or end users should not depend on installation details, we introduce an abstraction layer based on a simple semantic domain model. Furthermore, more devices connected to the home result in more diverse groups of users and services that interact with smart home solutions. This requires a new smart home access control concept. For example, the smart home system should not allow a utility company to

access the home owner’s health-related data from medical devices. To satisfy this requirement, we propose a new access control model and its implementation supporting our policy model for different users, permissions and multi-attributes including user roles, device type, location, device status and time.

This chapter describes our holistic and extensible software architectural framework called *Integrify* [99, 98] that seamlessly integrates heterogeneous protocol- and vendor-specific devices and services, while making these services securely available over the Internet. The architectural framework is developed on top of the OSGi framework and incorporates a semantic model of a smart home system. As a result, *Integrify* achieves semantic interoperability – the ability to integrate new applications and device drivers into the deployed system during runtime. Furthermore, it provides a new access control model for specific smart home scenarios.

The remaining of this chapter provides requirements for smart home architecture in Section 3.1, our proposed smart home gateway architecture including device stack, discovery of devices, message framework, and the semantic integration in Section 3.2. We present the access control mechanism in Section 3.3, and proof of concept in Section 3.4.

3.1 REQUIREMENTS FOR SMART HOME ARCHITECTURE

A smart home user study [53] uncovered that high cost of ownership, inflexibility, poor manageability and difficulties in achieving security constitute four barriers for broad adoption of home automation. We find that the heterogeneous nature of the smart home system as well as the diverse group of users is an intrinsic cause of these barriers.

Integrify is developed based on the in-depth analysis of the requirements of the different groups of users (stakeholders). The stakeholders considered in the system include the diverse roles of users in the smart home system as well as application developers using the Integrify’s APIs.

Plug-and-play of new devices during runtime

Homeowners often add home devices incrementally over time due to limited budgets and innovations in the market. Therefore, the combined process of seamless plug and play of devices and discovery of semantic services is necessary to allow flexibility in managing smart home systems. In our scenario, a home owner brings a new device (e.g., an Insteon dimmer light) to the home. She uses her smart phone camera to scan a barcode of the device, which is used by the smart home system to provide a corresponding discovery wizard for the device installation. The smart home system connects to the application store in the cloud in order to download relevant software drivers and basic applications recommended by the system. Upon the home owner's approval, the software can be deployed on the smart home system. Once this discovery and installation process is completed, the home owner is able to access the discovered device remotely using her smart phones or other user interface devices.

Supporting application development for 3rd party developers

The smart home applications should be developed without detailed knowledge about devices and protocol specific information. Instead, the smart home system should allow the applications to rely on an inference capability. Assume that a developer wants to develop an application to provide a service to turn off all lights on the first floor of a smart home if there is no one present for more than a certain period of time. Since every home has different device types and a different configuration of devices, the developer should be able to get a list of available devices with a certain level of abstraction. For example, the developer would request a list of the available lighting devices using the abstract interface and get the corresponding device states to determine the occupancy status without knowing the device address, protocol and other configurations. Thereafter, the third party developer can command the appropriate devices to carry out a certain action; in this case, after checking the state of the motion detection sensors, the program would shut off all lights on the first floor.

Access control for different types of users

The smart home system connects many different devices, which can be used by diverse

users. The types of users include adults and children in family, visitors such as babysitters, and service providers (e.g., utility company personnel or equipment maintenance contractors) accessing smart home remotely in form of services. This requires a flexible yet fine-grained access control mechanism to securely manage various smart homes with different configuration of devices and stakeholders. Our scenarios include access control that utilizes device status, device location, and current time as access control attributes.

3.2 SMART HOME ARCHITECTURE

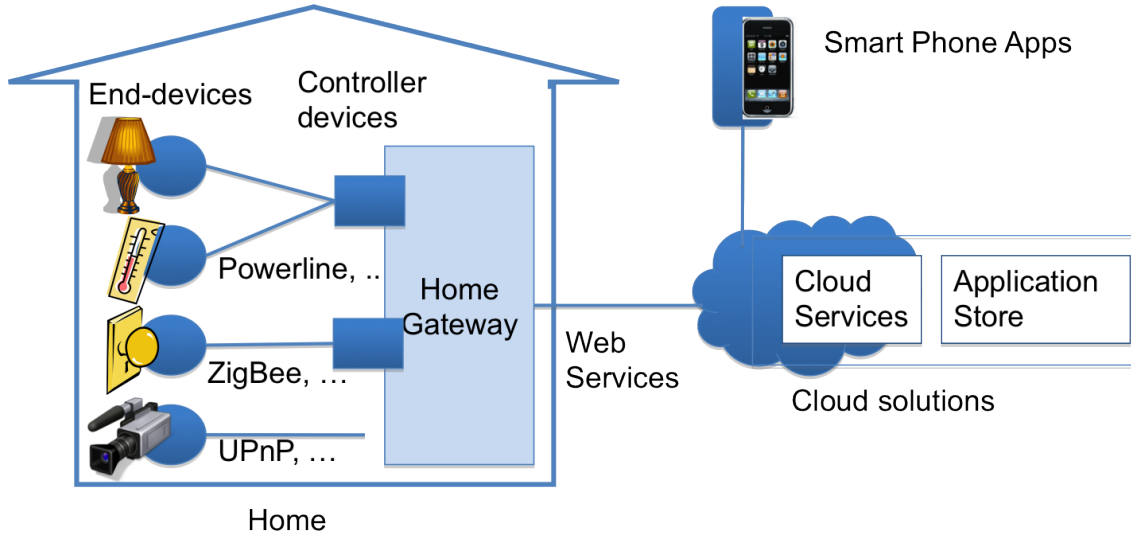


Figure 1: High level architecture for smart home systems

The *Integrify* system, illustrated in Figure 1, consists of a *home gateway*, which connects different types of home devices and provides unified interfaces that are accessible through REST web services [144] by using smart phones (our development focus is smart phones, although computer web access is trivially done). Home devices are categorized into *end devices* and *controller devices*. *End devices* are sensors and/or actuators, such as temperature sensors, video cameras, smart appliances, plug-in modules or any device that provide some

direct smart functionality. *Controller devices* do not offer specific services within the home, but are instead processors and processes (i.e., gateway-type devices) that allow the *home gateway* to communicate with *end devices*. The *home gateway* is also a processor hub with a specific set of processes that connect to the *cloud solutions* component outside the smart home; in particular, as the name suggests, the home gateway can connect with services that reside in the cloud to facilitate access. In particular, cloud solutions encompass an *application store* and provides *cloud services*. The *application store* manages the dependencies of software and home devices, and downloads software drivers and applications at the bequest of the user's *home gateway*. By using services that reside in the cloud, *Cloud services* are able to offer software services that require large amounts of storage or computation, such as a video surveillance service and an energy optimization service with machine learning algorithms.

Clearly, the system should not rely on an internet connection, due to the reliability of such services. the home gateway is autonomous enough to control all the devices internally. For that, the home gateway and the cloud services synchronize periodically to keep the state (e.g., schedules and access control for each device, but not the sensor or derived data) and has a very simple maintenance algorithm (i.e., much simpler than the machine learning algorithms used in the cloud solutions).

3.2.1 Smart Home Gateway Architecture

Important architectural requirements for our smart home system are *interoperability* and *dynamic integration* of many types of drivers and devices. As described in Section 3.2.1.3, different types of devices and services can be added and removed during runtime, and services within the system need to discover the existence of other services with which they need to interact. To tackle this problem, *Integrify* is built on top of the OSGi framework, which enables software bundles to be plugged in and out at runtime without shutting down the system.

Figure 2 illustrates the software building blocks of our smart home gateway. Our architecture employs a device stack and a dynamic message-handling framework (*Message FW*)

that processes commands in different ways depending on the context. The *HIM* (Hardware Interface Manager) detects any controller device that is connected to the system hardware. The *Deployment Manager* connects to the application store in the cloud and forwards to it properties of controller devices discovered by the *HIM* in order to download the proper drivers for new devices. It manages installations of software and updates of the home gateway profile in the cloud. *Access Control* manages authentication and authorization for different requests. Remote access is available through *REST APIs*.

3.2.1.1 Background of OSGi The OSGi Alliance, formerly known as the Open Services Gateway initiative (OSGi) is an open standards organization for specifications that enable the modular assembly of software built in Java. The OSGi specification includes a framework, resembling an embedded application server, for dynamically loading and managing software components, called bundles. These bundles can be dynamically instantiated by the framework to implement particular services [22, 116].

Device access service allows multiple devices to be discovered and their services advertised by the framework so that they can be made available to other devices, services, and applications [70]. *Integrify* uses this specification when we develop a plug-and-play of various devices with different protocols.

Event admin service defines a general inter-bundle communication mechanism. The communication conforms to the popular publish/subscribe paradigm and can be performed in a synchronous or asynchronous manner. *Integrify* uses this service by creating our own events for the communication between different components within the system.

3.2.1.2 Home Network Communication Technologies Prevailing home network devices are extremely heterogeneous: there are several standards and devices use different communication media (such as powerline and various RF bands), device addressing schemes (static or dynamic addresses), and different device discovery mechanisms.

There are extreme differences in the amount and protocol for devices to provide its capabilities (device description) to the system. Low data rate protocols such as X10 [147] and Insteon [62] do not provide device descriptions. ZigBee [24], KNX [26], Bluetooth [43]

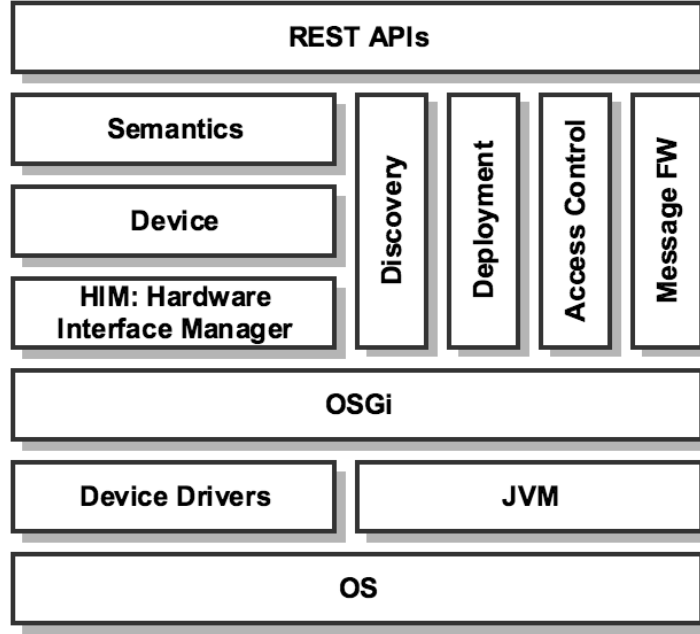


Figure 2: Core software building blocks for the home gateway

and EnOcean[1] provide device descriptions in the form of profiles specific to domains, such as home automation. The UPnP [138] and DPWS [4] standards use XML to describe device information. A common lack in all of these is that semantic descriptions are not provided by any of these home networks.

Security mechanisms of these protocols vary in behavior. Examples are pairing mechanisms of devices, checksums of message payload and data encryption techniques using symmetric keys. Due to these diverse and often incompatible mechanisms from different network standards, smart home systems in the market remain fragmented and provide only partial solutions addressing single protocols and subsets of devices.

The users of end devices at the smart home, however, are not interested in underlying communication protocols. For the smart home to fully realize the benefits of interconnectivity, devices must be able to discover and communicate with each other to provide compound services across these different home network protocols and technologies [70].

3.2.1.3 Smart Home Device Stack and Discovery Our smart home device stack in Figure 3(a) allows for uniform services to access diverse types of devices. *SH Service* (*SmartHomeService*) represents a service provided by a device in a protocol-agnostic way. *SH Device* (*SmartHomeDevice*) represents a proxy to a physical end device. This object contains common home device information such as the device’s URI (Universal Resource Identifier), device type (e.g., video camera), services (e.g., streaming the video) supported by this device, protocol information and device location. *Service Adaptor* is specific to a protocol implementation for *SH Service*, that is able to connect the *SHDevice* with the *Controller Driver*. The *Controller Driver* represents the features available by the controller device, including the low-level handling of commands. *Controller Driver* is responsible for acting as the base driver or for communicating with lower-level hardware drivers, such as an implementation of the Universal Serial Bus (USB) protocol to send and receive information from the device.

Figure 3(b) shows an example of the device stack. The Insteon device’s two capabilities of turning on/off and dimming the lamp are represented as two protocol-agnostic services of *OnOffService* and *DimmerLampService*. *InsteonOnOffServiceAdaptor* and *InsteonDimmerLampServiceAdaptor* are Insteon protocol-specific implementations to turn on/off and dim the lamp device. *InsteonPowerLinc Controller Driver* is an implementation of the *Insteon Controller Driver*, which sends commands to the *Physical Insteon Controller Device*. Then the *Physical Insteon Controller Device* broadcasts commands to all *Physical Insteon End Devices* to control the Lamp. Clearly, although Insteon uses broadcast, only the *End Device* with the same device Id in the command will act upon receiving the command.

New Controller Discovery

The controller discovery is described through an example illustrated in Figure 4. Upon plugging in a controller device (such as the PowerLinc RS232), the *HIM* detects a new controller and sends controller device information to the *Device Factory* module. The *Device Factory* module, which is responsible for the life-cycle of devices, creates a *Controller Device* with device information, and registers a new device event.

The *OSGi Device Manager* detects this event and starts the matching process of *Con-*

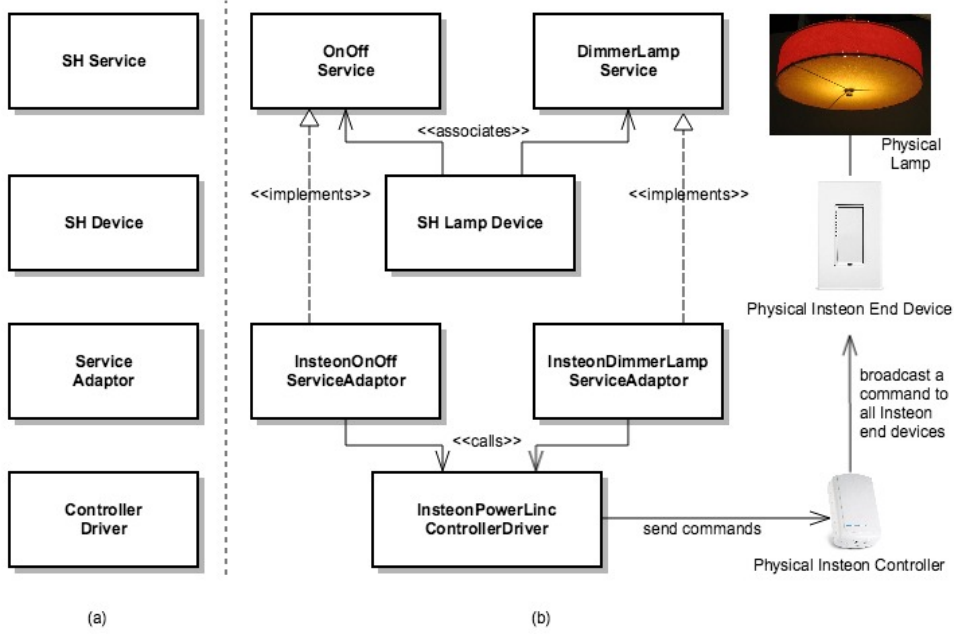


Figure 3: Device stack (a, left) and an instance of Insteon device (b, right)

troller Device and the corresponding driver (*Controller Driver*). If no driver is found, the *OSGi Device Manager* contacts *Driver Locator*, an *OSGi Service* interface for locating drivers for a device. Our implementation of *Driver Locator* calls *Deployment Manager*, which is responsible for downloading and installing the required software bundles from the application store in the cloud.

The *Application Store* manages software dependencies of the connected devices and other properties such as cost of software packages. Once the installation is complete, *Device Driver* registers a *Controller Driver* and the *Device Manager* restarts the matching process; the *Controller Device* finally matches with *Controller Driver*. With this approach, any new protocol-specific gateway type devices, which is called as *Controller Devices*, can be registered during runtime dynamically.

New End Device Discovery

Each protocol has its own device discovery mechanism. Discovery mechanisms are categorised into three types: *manual*, *semi-automatic* and *automatic device discovery*.

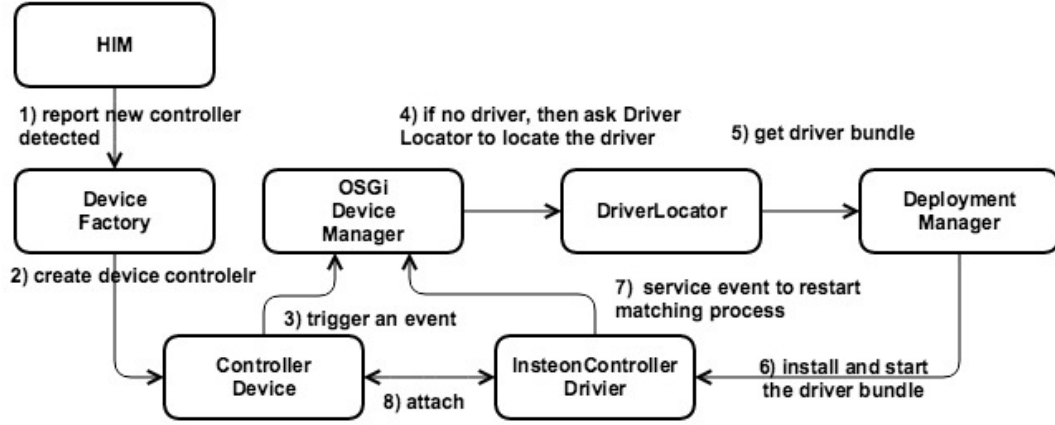


Figure 4: New controller discovery process for the unknown controller driver to the gateway

Manual discovery is device discovery that must be done entirely outside of the protocol specification. One example is X10, which provides no means for automated discovery of devices. In these instances the home gateway must derive all information about the device via user input.

Semi-automatic discovery applies to protocols that support some level of device discovery using the protocol, but still require human involvement. Insteon is an example of *semi-automatic discovery* in that the user is required to either 1) input the hardware address of a device to the controller device, or 2) press a hardware button on the device to initiate a discovery mode. The ZigBee protocol also falls into this category of home automation profiles.

Automatic discovery does not require any user interaction unless the system wants to get additional (non-protocol) information specific to the user's environment. An example of *automatic discovery* is the UPnP protocol, which detects and adds new devices automatically to the network system. In *Integrify* system, for security purposes, users are asked to confirm the automatically discovered device and ask them to provide semantic information such as location (room name) and a user-friendly name for the discovered device.

Manual discovery and *semi-automatic discovery* rely on user interaction to add new

devices to the smart home system. To mitigate the complexity of the discovery process, our architecture utilizes a camera on a smart phone device. Our smart phone application is capable of scanning the barcode of devices and sending the barcode to the application store, which manages the discovery parameters specific to the device manufacturer’s model. Depending on the discovery parameters returned by the application store, users are guided with a corresponding wizard to easily add the new device.

At the home gateway, we introduce *Device Discoverer* modules for each protocol. Depending on the input from the smart phone, the home gateway calls the corresponding discoverer to add the new device. Figure 5 illustrates an example of new ZigBee end device discovery.

In our scenario, the user does not know the discovery mechanism that ZigBee supports. The user scans the barcode of the ZigBee end device with her smart phone camera using our smart phone application. The smart phone application sends a request to *Application Store* in order to get the discovery parameters for the scanned barcode data. Upon the discovery parameters, the smart phone application provides the proper wizards to discover a new device and then sends the discovery request with parameters to the home gateway. The home gateway finds the corresponding discoverer and sends a broadcast message to the local ZigBee network through *ZigBeeController Driver*. This is equivalent to pushing a button on the controller.

Once a new device is discovered, similar to the controller device discovery process, the *Device Factory* component creates a new device object (*SH Device*) and the *OSGi Device Manager* matches the corresponding *Service Adaptors* for that device. *Service Adaptors*, in addition to matching to *SH Device* object instances, must be assigned to a *Controller Driver* in order to execute commands.

3.2.2 Message Framework

We introduce a smart home internal message (*SH Message*) to communicate with different software modules in the home gateway. *SH Message* can represent various types of messages, such as command invocation messages (e.g., turn on kitchen lights), or state change messages

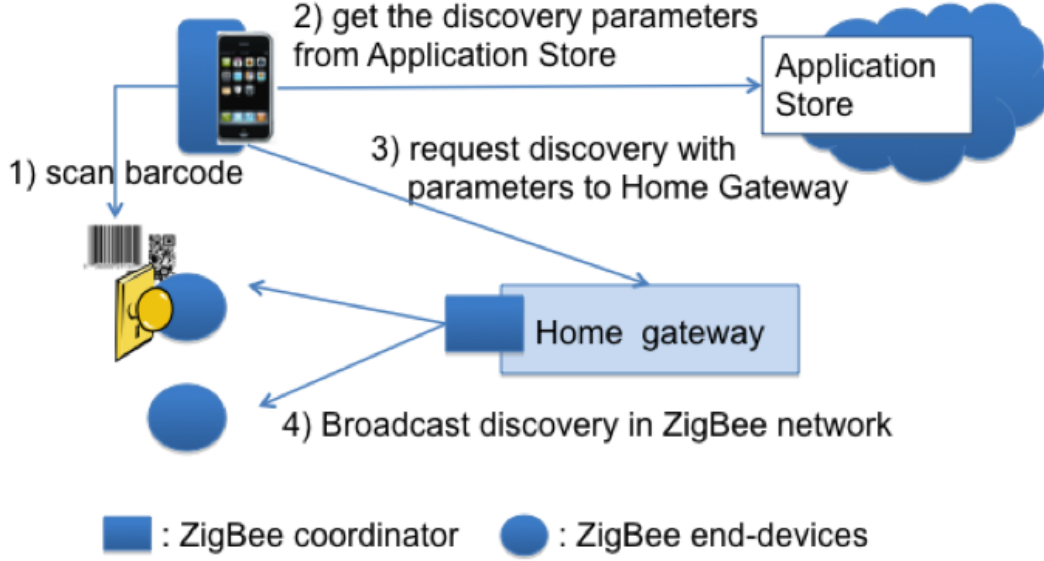


Figure 5: ZigBee end device discovery process

(e.g., data updates from a motion sensor if it detects motion).

SH Messages are handled in a centralized yet dynamic way using a mixture of *Chain of Responsibility* and *Strategy* design patterns [78]. A *SH Message* is handled by invoking a list of *Message Handlers*, each of which is a logical entity responsible for executing a specific task such as raising an event throughout the system or logging the message. The list of *Message Handlers* is constructed on-the-fly by the *Chain Factory* module, which uses the message type to derive the purpose of the *SH Message* and how it should be handled. In this way we provide an extensible framework where handlers may be added and removed dynamically.

Figure 6 illustrates how our message framework works for an example service. A Web service creates a *SH Message* with priority and timestamp to invoke an action on a device (in this example setting the temperature value) and inserts it in the queue. The *Chain Factory* then constructs the chain of *Message Handlers* that will handle the message. In this example, *Event Publisher Message Handler* raises a new event to any modules that are interested in being notified of this particular type of occurrence (the set temperature command). Following this, the *Log Message Handler* handles the message, which will record the execution of the

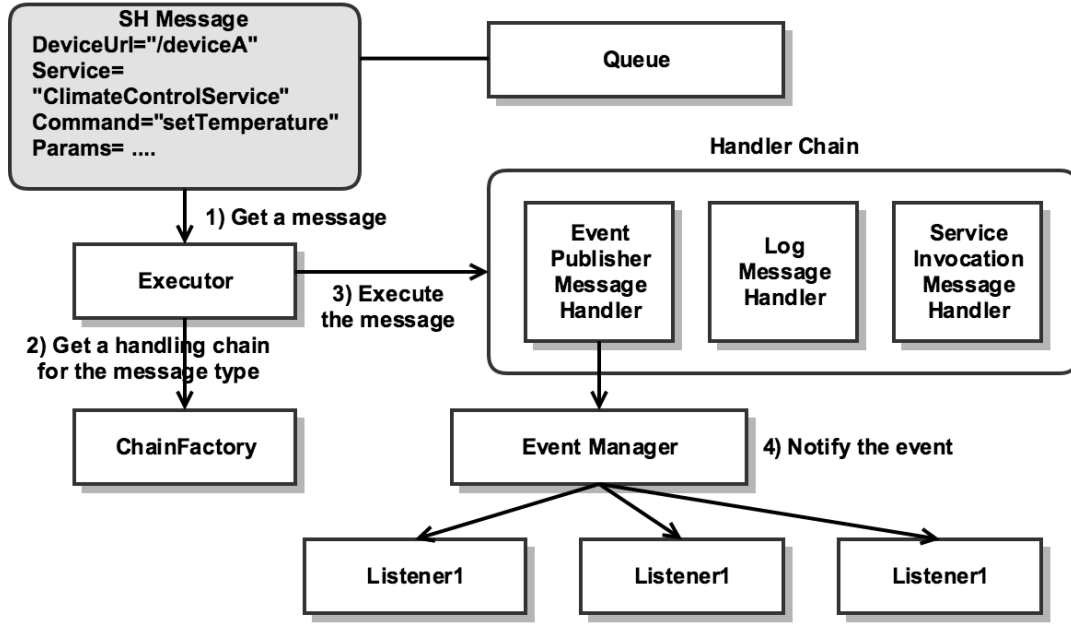


Figure 6: *Integrify* message framework

message in the log. Finally, the *Service Invocation Message Handler* carries out the message’s purpose by sending the message’s intended command to the appropriate *Controller Driver*.

3.2.3 Semantic Integration

Our architecture incorporates a semantic model that enables new applications to be developed independently from the concrete environment in which they are deployed (see Section 3.1). The semantic model is formalized as an ontology. Ontologies translate a domain of interest into a set of concepts, properties and relations governed by strictly formalized semantics. Automatic inference operations can draw implicit conclusions from the explicitly stated knowledge. Consider a typical yet motivating example application such as “turn on all lights on the first floor.” The application will use an abstract lighting super-type to retrieve all possible lamp devices. Moreover, based on the specified device location and the contextual knowledge (e.g., kitchen is on the first floor), it has to infer that devices located

in the kitchen are also located on the first floor. After analyzing such and other examples, the following elements are considered essential for the semantic knowledge in the smart home domain and its potential use by new applications:

- Taxonomy of home devices and home environment to allow for abstraction and on-the-fly semantic device retrieval, together with contextual knowledge (e.g., location and time) required for context-aware applications.
- Reasoning about the implicit knowledge from a minimum number of explicit facts.
- APIs to retrieve and modify semantic knowledge without in-depth understanding of formal ontology descriptions.

Knowledge about the smart home system has been addressed by domain ontologies such as DogOnt [45] and Hydra [73]. The semantic model for the *Integrify* system is built on top of DogOnt. Figure 7 shows an excerpt from our adapted domain ontology that supports the example application presented above: transitive location properties (*isIn* or *contains*) enable contextual reasoning. The concrete lamp device instance is grounded in the multi-level device taxonomy and connected to the services it offers, which allows for abstract semantic device retrieval.

We formalized the ontology using the Web Ontology Language (OWL) [34], because it is well supported by toolkits such as Apache Jena framework [92] and reasoning engines such as Pellet [154]. Note that the background of Semantic Web is provided in Section 4.5.1. The related work in smart home semantic models and reasoning engines is discussed in Section 2.5 and Section 2.2 respectively.

3.2.3.1 Reasoning Engines Real-time smart home applications require fast responses, making efficient reasoning engines especially important. High performance reasoning can be achieved by excluding certain constructions from OWL and thereby reducing the expressivity of the model. However, some of those constructions are important to automatically detect an inconsistent model that might indicate a configuration mistake. One example is disjointness of device classes, which prevents a device from being simultaneously configured as two different and conflicting device types (e.g., as a lamp and as a TV).

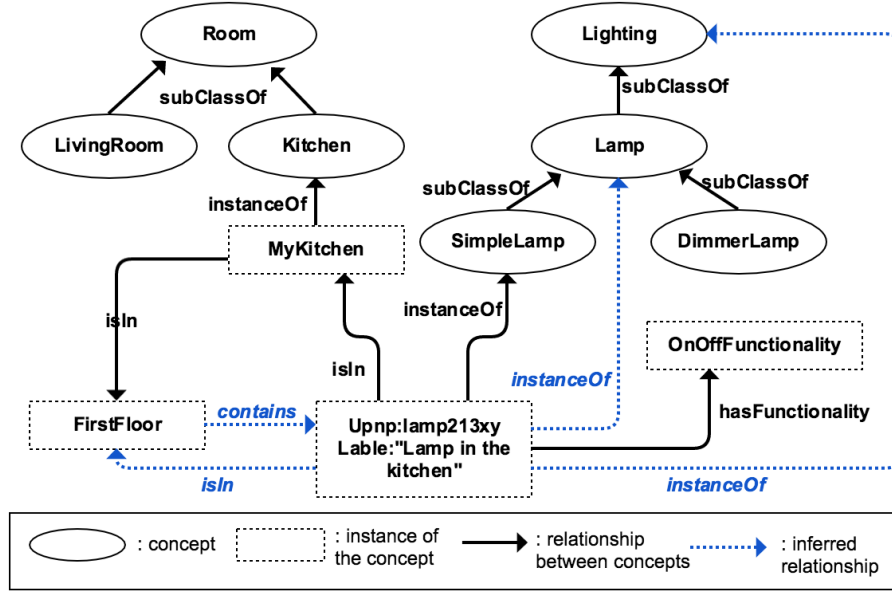


Figure 7: Excerpt of the ontology representation

We distinguish *online* use cases such as semantic device retrieval by applications that requires fast responses, and *offline* use cases such as consistency checking and device instance classification during the gateway initialization where low response times can be traded in for higher expressivity. Depending on the use case, we employ a suitable reasoning engine.

Online reasoning along with data storage is handled by OWLIM [102], a fast and scalable reasoning engine that supports full RDFS semantics and parts of OWL-DL. For offline reasoning use cases, the more powerful Pellet [154] reasoning engine guarantees the consistency of the model and fully classified device instances.

We run both *online* and *offline* reasoners on the same knowledge base although they are built on top of slightly different RDF frameworks: While Pellet uses the Jena framework [61], OWLIM is based on the Sesame framework [52]. We designed our system architecture to include an adaptor that can transfer RDF triples from Jena/Pellet to Sesame/OWLIM and vice versa. We attempt to quantify our expectations about the trade-off between expressiveness and response time, as well as the runtime impact and the scalability of the semantic

model in an experiment. The detailed results and our analysis can be found in Section 3.4.2.

3.3 ACCESS CONTROL FOR SMART HOMES

This section discusses how *Integrify* supports the flexible yet fine-grained access control for the smart homes with different roles of users and devices. We propose a new policy model that utilizes device status, device location and current time as access control attributes for the smart home system. We realize the proposed new policy model in our *Integrify* framework by combining 1) OSGi User Admin service for authentication and user’s roles retrieval and 2) eXtensible Access Control Markup Language (XACML) [124] for access control supporting the attributes mentioned above.

3.3.1 Policy Model

Our policy model includes different *roles* (e.g., administrator, adult, kid), and it allows a user to be a member with a variety of roles. The policy model has four *permission types*: user management, device management, controlling device, and monitoring device. *User management permission* represents the ability to add, remove and modify user roles. *Device management permission* represents the ability to add, remove and modify devices. *Controlling device permission* represents the ability to issue a command to a device such as opening a door lock or turning on a light. *Monitoring device permission* represents the ability to get state information from sensor devices, which includes getting temperature data, getting heart monitor data and video streaming.

Our smart home policy model represents fine-grained access policies similar to the ones we enforce in real life. We identified four attributes capable of expressing a wide range of rules when combined. The *device type attribute* is important in the smart home domain, because our system interacts with different user roles (e.g., utility company, health-care provider) from outside the home with limited access to certain types of the devices. For example, smart meters can be accessible by utility companies but blood pressure monitoring devices

should not be accessible by utility companies. Also, a family usually consists of different age groups that often require different access policies for different types of devices for safety or other reasons. The *location attribute* is useful for more fine-grained control. A house resident can limit the access of some places within the home for visitors. For example, a visitor may be allowed to control devices in the living room, but not in the bedroom. The *device status attribute* is also required especially concerning future smart grid integration. One possible scenario in the future smart grid is that the service provider should not turn off the laundry machine while the machine is on. The *time attribute* provides users with a flexible way to control access during different times of the day, such as common parental control schemes. For example, parents may only allow their kids to access entertainment devices (e.g., television) during specified time durations.

3.3.2 Access Control Architecture and Design

To realize access control within our architecture, we develop a hybrid approach of *OSGi User Admin service* and *eXtensible Access Control Markup Language (XACML)* [124]. While the *OSGi User Admin service* is moderately expressive, it does not allow us to express the multitude of variables introduced in our policy model. Thus, we use it only to represent subjects (user roles) and assist in our authentication process, while the more expressive *XACML* is used to specify and enforce the policy for given subjects.

Figure 8 illustrates how access control concepts are implemented in our architecture. The client request is transformed into new internal messages (*SHMessage* discussed in Section 3.2.2). Each message is wrapped with an authentication header, which contains a flexible form of user credentials such as username/password or an authentication token. The message is then enqueued within the message framework for execution. The *AuthenticationHandler* consults the *OSGi User Admin service* to validate the given credentials and get the user's roles. The *AuthorizationHandler* fulfills the role of *Policy Enforcement Point (PEP)* as specified by *XACML*. It checks the authorization object of the message (representing the subject), the intended action and the object of the message, as well as other attributes such as location and time, and submits an *XACML* query based on this information to the *Policy*

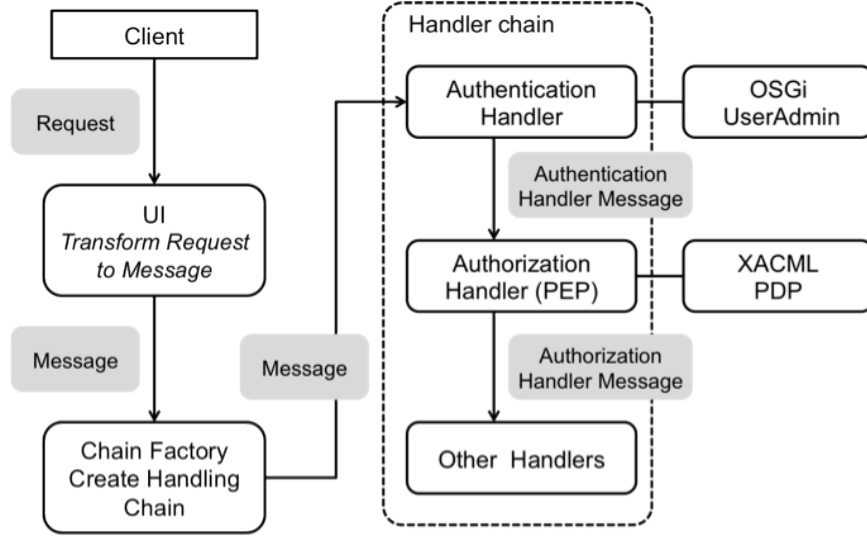


Figure 8: Access control design concept

Decision Point (PDP). The *PDP* evaluates the *XACML* policy to see if the intended action (e.g., turn on an entertainment device) is indeed an authorized one. If the policy allows the request, the *PDP* will refer the message to the next handler for further processing. If the policy does not permit the action, the message is denied and dropped. Code 1 is an example of the *XACML* policy used in our demonstrator, “permitting kids to turn on entertainment devices only before 7 PM.” Note that this policy permits kids to turn off entertainment devices anytime because the policy contains only “on” status of the device status attribute.

3.4 INTEGRIFY IMPLEMENTATION: PROOF OF CONCEPT

This section provides how the proposed *Integrify* software architecture and access control mechanism are realized in a real system. It also discusses our evaluation results of the real-time behavior of the prototype system with different configuration of semantic reasoning engines.

Code 1 Example of XACML policy (shortened): Permit kids to turn on entertainment devices only before 7 PM

```
1 <Rule RuleId="Rule_6" Effect="Permit">
2   <Target>
3     <Subjects>
4       <Subject>
5         <SubjectMatch MatchId="string-equal">
6           <SubjectAttributeDesignator DataType="string" AttributeId="group"/>
7           <AttributeValue DataType="string">Kids</AttributeValue>
8         </SubjectMatch>
9       </Subject>
10    </Subjects>
11    <Resources>
12      <Resource>
13        <ResourceMatch MatchId="string-equal">
14          <ResourceAttributeDesignator dataType="string" AttributeId="domain"/>
15          <AttributeValue DataType="string">Entertainment</AttributeValue>
16        </ResourceMatch>
17        <ResourceMatch MatchId="string-equal">
18          <ResourceAttributeDesignator DataType="string" AttributeId="status"/>
19          <AttributeValue DataType="string">On</AttributeValue>
20        </ResourceMatch>
21      </Resource>
22    </Resources>
23    <Actions>
24      <Action>
25        <ActionMatch MatchId="string-equal">
26          <ActionAttributeDesignator DataType="string" AttributeId="action-id"/>
27          <AttributeValue DataType="string">control-device</AttributeValue>
28        </ActionMatch>
29      </Action>
30    </Actions>
31  </Target>
32  <Condition FunctionId="time-less-than-or-equal">
33    <Apply FunctionId="time-one-and-only">
34      <EnvironmentAttributeDesignator DataType="time" AttributeId="current-time"
35      </Apply>
36      <AttributeValue DataType="time">19:00:00</AttributeValue>
37    </Condition>
38  </Rule>
```

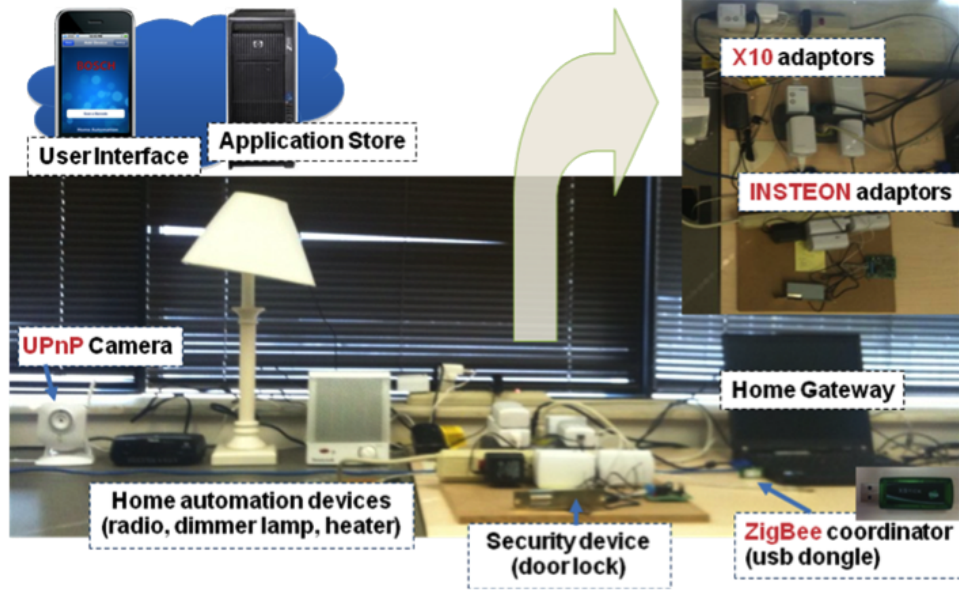


Figure 9: *Integrify* demonstrator

3.4.1 Implementation

We demonstrate our smart home architecture and design by implementing a prototype that consists of an actual home gateway, real devices, a private cloud and a smart phone as a user interface. The prototype realizes discovery of selected new devices and integrates services based on semantic information. Our access control concept is also demonstrated with example policies covering selected users, devices and services.

Figure 9 is a picture of the prototype system that realizes the *Integrify concept*. The home gateway prototype is running on Ubuntu OS in a laptop, supporting two OSGi implementations of ProSyst [16] and Equinox [118]. Prosyst is a commercial software of OSGi implementation and it provides tools that help develop the *Integrify* system easily. Equinox is an open source software, thus it does not require any cost to develop the system. The prototype system integrates devices using X10, Insteon, ZigBee and UPnP protocols. We have integrated various different device types: sensors (e.g., motion, water leak), on/off adapters for home appliances and dimmer lamp adapters, RF transmitters to control TV, and video surveillance cameras. The user interface devices run on iOS and Android devices as native

apps. The cloud web solutions currently include the application store component, which is deployed in our private cloud.

During runtime, the user connects X10, Insteon and ZigBee controller devices via USB to the home gateway. The home gateway then detects the presence of new controller devices and creates new controller device objects. The UPnP drivers automatically discover a new UPnP video camera when it is plugged into the network and send a notification to the user’s smart phone (automatic discovery). For other end devices, our demonstrator uses iOS and Android applications to start discovery of end devices by scanning the barcode of the devices. Note that the different device discovery mechanisms for automatic, semi-automatic and manual discovery are discussed in Section 3.2.1.3. We demonstrate the message framework and the semantic abstraction layer by implementing a demonstrator via a couple of applications, namely simple remote control of devices and an occupancy dependent lamp control. The demonstrator also contains a policy model to show access control for the different users, permissions and attributes discussed in Section 3.3. Users can remotely access discovered devices using smart phones based on the access control policy. Our example policy grants all permissions to the users having the adult role, while it restricts accesses for the users belonging to the kid role. Example policies include “kids are allowed to control all lighting devices in the guest room.” and “kids are allowed to turn on entertainment devices before 7pm.” We use a *permit-override* algorithm, allowing a single evaluation of permit to take precedence over any number of deny. In other words, if a policy set contains multiple policies and those policies return different decisions including one *permit* and multiple *deny* for a given request, then the permit is applied to a given request.

A demo video for the *Integrify* prototype system is available in [96], which uses the devices, gateway, smart phones and application store shown in Figure 9. The video first shows the remote access and control of heterogeneous devices through an iOS native application, for example by turning on/off a lamp registered to the system. The next scene in the video shows the plug and play of controller device: it first shows that there is no notification on the *Integrify* iOS application; then the user plugs in a X10 controller device to the home gateway through a USB port. The home gateway asks the application store with the controller device information obtained from USB and downloads a software bundle for the controller device

driver. The smart phone application gets a new notification from the home gateway, which asks a user to install a new controller device driver (X10/PLC controller driver). Once the user clicks an *install* button on the iOS application, then the X10 controller device is added to the home gateway. The next scene in the video shows a new end device discovery with a manual discovery mechanism. The user brings a new radio, which is connected to an X10 On/Off adapter. The user clicks the profile icon on the iOS app, and click the plus button to add a new device. The wizard on the iOS application guides a user to add device name, device address, device location and device protocol. Once the device is successfully added, the iOS application screen is back to the main menu. The user clicks the entertainment icon in the main menu, then the screen shows the list of the entertainment devices and the user can remotely control the radio device by clicking the turning on/off button on the iOS app. The last scene shows the remote access and control of lamps and radios through both iOS app and Android app.

3.4.2 Evaluation

To evaluate the real-time behavior of our prototype, especially regarding the overhead introduced by reasoning on a semantic model, we measure the difference between the time the gateway receives a service call from the remote user interface and the time it sends the command out to the controller device. The execution time includes only time spent in the gateway. It excludes the delay caused by network communications, in particular the remote call to the gateway and the device protocol overhead. We compare the execution time for different numbers of devices ($n = 1, 5, 20, 50$) and three different settings: 1) *Jena*: semantic retrieval of the device given a particular location using Jena/Pellet for ontology access, 2) *Sesame*: the same scenario but with Sesame/OWLIM, and 3) *Nosemantic*: direct access to a lamp by its identifier without semantic retrieval. For each experiment, we restart the gateway and send 50 remote service calls (i.e., turn on and off a lamp device) for each n devices.

Our hypothesis is that the system should be scalable with constant execution time regardless of the number of devices discovered. This means the average execution time is not

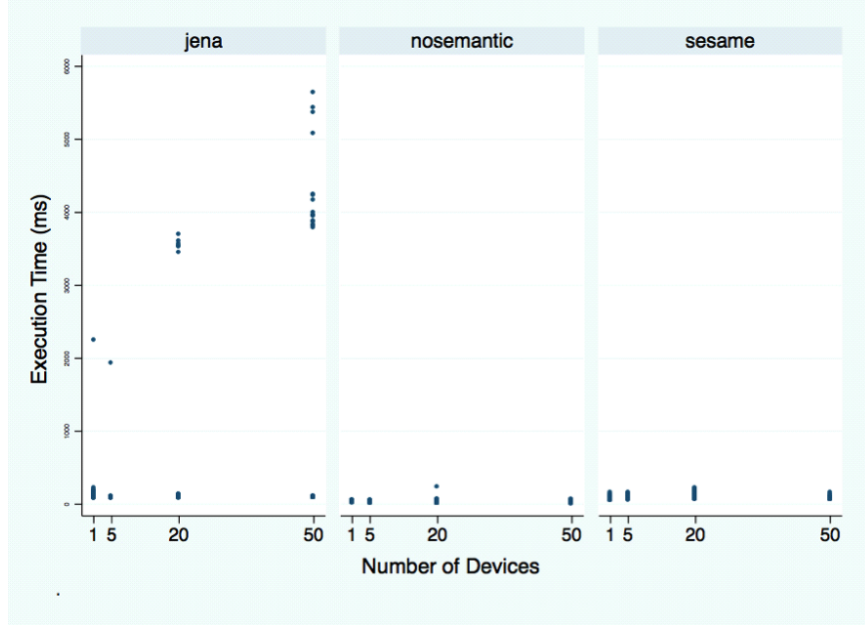


Figure 10: Execution time of service calls for the different experimental settings

Configuration	Mean	Std	Min	Max
Jena	558.33	1286.65	83	5641
Nosemantic	25.45	20.34	4	235
Sesame	89.67	34.44	52	215

Table 1: Statistics for execution time (ms) of service calls to lamp device

influenced by the number of devices, and the overhead of the semantic device retrieval should stay constant for a growing number of devices. Our working hypothesis is that the average execution time of *Sesame* (used for on-the-fly semantic access, low response times) will be lower than that of *Jena* (static consistency checking, higher expressivity).

Figure 10 gives an overview of the execution times of all service calls for the different experimental settings, while Table 1 presents the statistical results for the execution time

of service calls for the different experimental settings. For *Nosemantic* and *Sesame*, we observe no user-perceived significant differences in the distribution of the execution time for different device numbers. Although the average execution time with *Sesame* is 256% higher (overhead of the semantic device retrieval) than without semantic support, the variation is relatively low, specially when considering the IoT environments where human perception is not affected in the millisecond granularity. When the semantic services are provided by *Jena*, execution time is much higher on average. Also, the execution time varies tremendously over the number of service calls. This is mainly due to the first few service calls: After 1-10 calls a query caching mechanism sets in and reduces the delay for further calls to a level comparable to *Sesame*. The proportion of service calls with high execution time depends on the number of devices.

4.0 SOCIALITE: A CLOUD BASED DISTRIBUTED COLLABORATION FRAMEWORK FOR SOCIAL INTERNET OF THINGS

The Social Internet of Things (SIoT) is a new paradigm that merges the Internet of Things with Social Networks. In SIoT, all things can be socialized within a *new social network framework* established between people and things/devices. People and devices in social relationships will collaborate with each other by sharing and discovering new data (raw or processed) at scale. They make automated decisions by reasoning about new data and their knowledge.

Socialite [100] is a cloud based distributed collaboration framework to realize the vision of the SIoT. The *Socialite* architecture aims to provide *interoperability* of connected devices from different manufacturers participating in different ecosystems, *scalability* to handle large number of data streams from interactions between devices and people. We also expect that new device types that currently do not exist will be developed in the future. Therefore, the *Socialite* architecture is designed to support *extensibility* by taking future devices into consideration. Furthermore, *extensibility* greatly comes at the user's end when they are empowered to define and share their rules that consist of devices, services in various relationships flexibly for their interests.

This chapter discusses new social relationships for the SIoT and new types of applications in Section 4.1. A user survey on the SIoT for smart home systems is discussed in Section 4.2. The non-functional requirements for the *Socialite* system is discussed in Section 4.3. The *Socialite* system overview is explained in Section 4.4. The *Socialite* semantic models, which serves as a foundation for *interoperability* is explained in Section 4.5. The *Socialite* server architecture supporting *interoperability*, *scalability* and *extensibility* is discussed in Section 4.6. Section 4.7 provides an end user empowered reasoning framework including

the *Socialite* reasoning concept and the end user programming application, which is a part of the *Socialite* client application. Section 4.8 demonstrates how this realizes the proposed architecture through integration of real and virtual devices, services and user created rules to the system, and provides the evaluation of our architecture with respect to non-functional requirements.

4.1 NEW SOCIAL RELATIONSHIPS AND APPLICATIONS

The first insight of this work is realizing the necessity of new relationship types between people and devices. In this section, after defining these new relationships, we discuss motivating new types of applications coming from the proposed new relationships, which require a new software framework such as *Socialite*.

4.1.1 New Social Relationships

Socialite defines a set of new relationship types (see Table 2) that allows participants in Social Internet of Things to collaborate with each other, in addition to existing friendships among users in social networks. Table 2 summarizes the new social relationships in *Socialite* and Figure 11 illustrates these relationships with examples.

Users and devices participate in an *ownership* relationship if a user registers a device in *Socialite*. Users and devices have location information, and a *co-location relationship* between them is dynamically updated when their location is changed.

Socialite also provides explicit relationships between devices themselves, namely *thriendship*, *kinship* and *shared ownership*. We assign *kinship* for the same model of devices from the same manufacturer, and *thriendship* (things of friends) signifies the relationship between devices owned by friends. A user usually owns more than one device; therefore we introduce *shared ownerships* for the devices owned by the same owner.

The relationships in social networks represent which users can be related to other users. By ‘relate’, it means that two or more users have some form of association that leads them

Relationship type	Relationship definition
Friendship	Relationship between users, as in social networks
Ownership	A device registered by its owner
Co-location	Users and/or devices in the same location
Kinship	Devices with the same model and manufacturer
Thriendship	Friendship among things/devices of friends
Shared Ownership	Devices owned by the same user

Table 2: New proposed relationship types in Socialite

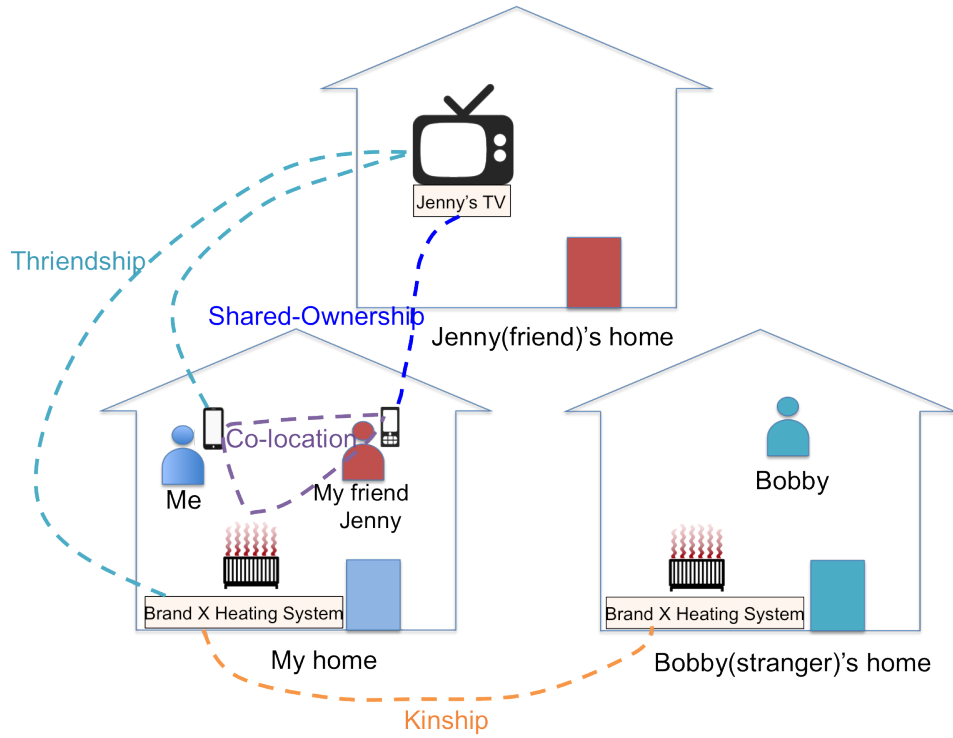


Figure 11: Graphical representation of the new social relationships in *Socialite*

to converse and share objects of sociality [95]. Consequently, how users of a social network platform are connected often determines the what-and-how of information exchange. The benefits of the *Socialite* system become larger when the new relationships are established for exchanging, distributing and receiving the data generated by the users and devices. The social capital can be increased in SIoT when the information is shared with participants in new social relationships and the discovered information are used by them.

4.1.2 New Applications Leveraging New Social Relationships

Below we present a few promising applications that show how users would benefit from adopting *Socialite* and its relationships for sharing information with other people and devices. To motivate the new relationships discussed in Section 4.1.1, we use three main applications in which information sharing among devices could be leveraged through the new relationships.

Device Life-cycle Management

Devices in the *kinship* relationship can use/discover new information such as parameter settings to improve their performance by sharing their configuration with other devices. This is particularly useful for less experienced users, who can allow their devices to adapt based on the information shared by more experienced users/devices (e.g., less tech-savvy users allowing their thermostats to learn from neighbors' thermostats). This actually happens more often than one would think; consider for now heating systems that could profit from this, that is, those installed in equivalent spaces (e.g., apartments on different floors, but in the same geographical position in an apartment building) would share the configuration parameters with other devices with relationships. These parameters may include air flow, target temperature, and other intelligent thermostat settings.

The *kinship* relationship also enables devices to share errors and repair history, such as an error generated by a boiler and replacement of spare parts or a service record for that particular error. When coupled with other information, *kinships* can also be used to predict a device failure. For example, the usage/load of a furnace together with the probability of the failure of that type of furnace, calculated based on the failure events from other similar devices in *kinships*, enable the owner of the same type of the furnace to receive predicted

information about future problems and schedule a maintenance service before the failure occurs.

Device Collaboration for Common Goals

The user's different devices, located in a common space (*co-location relationship*) and/or owned by the same user (*shared ownerships*) can share information among themselves and interact with each other to achieve a common goal. For instance, if a motion detector shares the presence information and a light sensor shares the light level information, then a lamp or a thermostat in the same room can adapt their settings based on the current information provided by other devices in relationships. Moreover, any applications that encompass intelligent rules or algorithms applicable to the historical data of the devices in relationships can also take advantage of such information. Another example is when a game encompassing both users and devices is created to achieve a goal collaboratively, for example, a friendly competition among neighbors can lead their appliances to consume less energy in a household.

Recommendation via Social Navigation

A common way to navigate an information space in the real world is with help of other people. This communication with other agents (human or artificial) to navigate an information space is called social navigation [71, 69, 126]. With the help of *Socialite*, it would be possible to provide recommendations from user to user, based on the friend's networks, same device types (*kinship*), and reputations. This would be done transparently, based on the request from users to their social networks in search of a device of a certain type with certain characteristics; for example, a user posts that s/he would like to purchase a new air conditioner or the air conditioner itself, knowing its lifetime is expiring, posts on behalf of the user, describing the characteristics of the current device: 12,000 BTU and top 10 energy efficient air conditioners in *thriendships*. Furthermore, the device itself can initiate the social navigation to inquire about the operational performance of devices in *thriendship*.

Discussion

It is possible to program our devices to achieve some of applications described above with other technologies and approaches. For example, a heating system manufacturer collects the data from all connected heating systems and proposes a new configuration of the

setting and/or parameters. However, this would be done in a fragmented way with current technologies, and therefore demonstrates the need to adopt a general framework such as *Socialite*, with the explicit representation of the different relationships for the humans and devices in the Internet of Things.

The *Socialite* framework can enable scalable and efficient ways of realizing the applications mentioned in this section. Since users and devices are able to query information, publish and give access to their own information, decision making is to be more collaborative.

4.2 USER SURVEY: SOCIAL INTERNET OF THINGS FOR SMART HOME SYSTEMS

We conducted a user survey in order to understand and identify the users' needs on the future smart home systems with our approach of the Social Internet of Things (SIoT). We investigated the users' desired features for the smart home system in SIoT specifically by asking example scenarios with new social relationship types in Section 4.1.1. We were also curious about their perceived acceptance on a hypothetical SIoT based smart home system, a user empowered rule creation, and sharing of their rules with other people. The participant was requested to answer these questions at the end of the survey along with demographic information.

This survey aims to use the analyzed survey results as a foundation for the development of the basic features, semantic models and the reasoning concept for the *Socialite* systems.

4.2.1 Methodology

The recruiting methods and the survey design are explained in this section.

Recruiting Methods: Participants to the survey were recruited in two ways: the Amazon Mechanical Turk (MTurk) [7] web site and the Facebook [2] network community of the author. MTurk is an online crowdsourcing labor marketplace where the registered crowd workers perform micro-tasks posted by the requester and get paid based on their submitted

results. As for the MTurk participants, a qualification was applied to the participants by restricting the survey to the US crowd workers who previously submitted more than 100 tasks and have greater than 95% approval rating. Each participant from MTurk was paid \$1.5 USD for approximately 30 minutes of the survey. 60 participants have participated in the survey through MTurk (55 participants) and Facebook (5 participants).

Survey Design: The survey questionnaires were designed to have four phases: 1) getting familiar with the smart home and SIoT concepts by watching two videos; 2) answering a user’s desire for automated features for smart home systems in general, which implies ownership relationships; 3) answering a user’s desire for automated features with consideration of new social relationships as explained in the Section 4.1.1; 4) answering a set of background questions and a participant’s perceived acceptance on the end user programming capability and sharing of their rules and the SIoT concept. All participants received the same questionnaire, which is hosted in Google Forms [11]. All participants were requested to answer the questions in 1), 2) and 3) in text fields except the background questions in 4).

In phase 2 of the survey, we asked the participant to watch two smart home promotion videos from LG [10] and Ericsson [5]. The first video introduces a general smart home concept without necessarily connecting to other smart homes. The second video leads the participants to the future with the *Social Internet of Things* paradigm because devices actively communicate and interact with each other as well as with the home owner. The participant is asked to provide one example from the video where two (or more) devices communicate or share information with each other. The participant is asked to provide two preferred applications from the videos. We ask this question to validate if the participant had a good understanding of the smart home and SIoT concept through the videos they watched and gained enough context to proceed with next questions.

In phase 2 of the survey, we attempt to obtain the features desired in a smart home. A picture of a smart home with various devices is presented to the participant to help him or her to come up with two smart home applications by using the devices shown in the picture. This phase focuses on only a single smart home, which implies an ownership relationship between the user and devices, if devices are used in their desired application description.

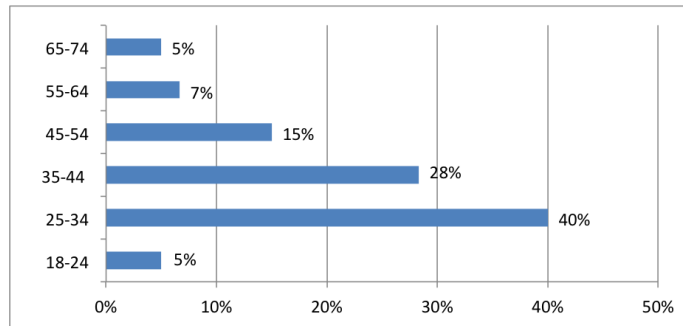
In the phase 3 of the survey, we attempt to obtain the features desired in a smart home

with consideration of the new social relationships. The new social relationships from the *Socialite* system including *thriendship*, *co-locationship*, *kinship*, and *shared ownership* are explained to the participant with examples of relationships using three homes: my home, a friend’s home and a stranger’s home. For this explanation, a visual description (see Figure 11) is used together with text based descriptions. The participant is requested to propose two automated applications leveraging each relationship.

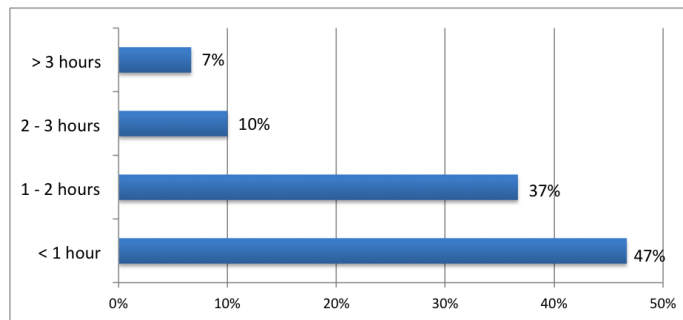
In the phase 4, the participants are requested to answer demographics, programming experience, social network usage, perceived acceptance of end user programming, sharing of their rules and a SIOT approach/concept similar to the *Socialite* system.

4.2.2 Demographics

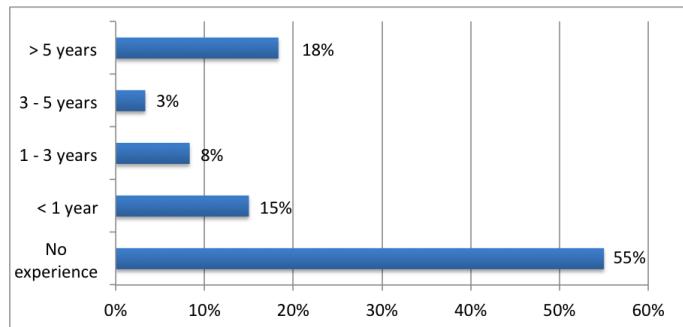
- *Age*: Our participants ranged in age from 18 to 74 (See Figure 12a). In detail, 40% of our participants were between 25 and 34 years old. 28% were between 35 and 44 years old, 15% were between 45 and 54, 7% were between 54 and 64 years old, 5% of them are each from between 18 and 24 years, and between 65 and 74 years old. (see Figure 12a)
- *Smart home experience*: 10% of the participants have smart home devices at their home.
- *Smart phone experience*: 93% of participants use smart phones.
- *Social networking site experience*: 98% of participants are currently registered to the social networking sites. 90% of participants have been using the social network sites more than 3 years. Participants use multiple devices to access their SNS accounts. 47% of them spend less than 1 hour per day in the social network site, while 37% spend between 1 and 2 hours, 10% spend between 2 to 3 hours and 7% spend more than 3 hours per day in the social network site (see Figure 12b).
- *Programming experience*: 55% of the participants do not have programming experience. 15% of participants have less than one year of programming experience, 8% of them have 1 to 3 years of programming experience, 3% of them have 3 to 5 years of experience, and 18% of them have more than 5 years of experience. (see Figure 12c)



(a) Age distribution



(b) Social networking service daily usage hours



(c) Programming experience

Figure 12: Demographics of participants

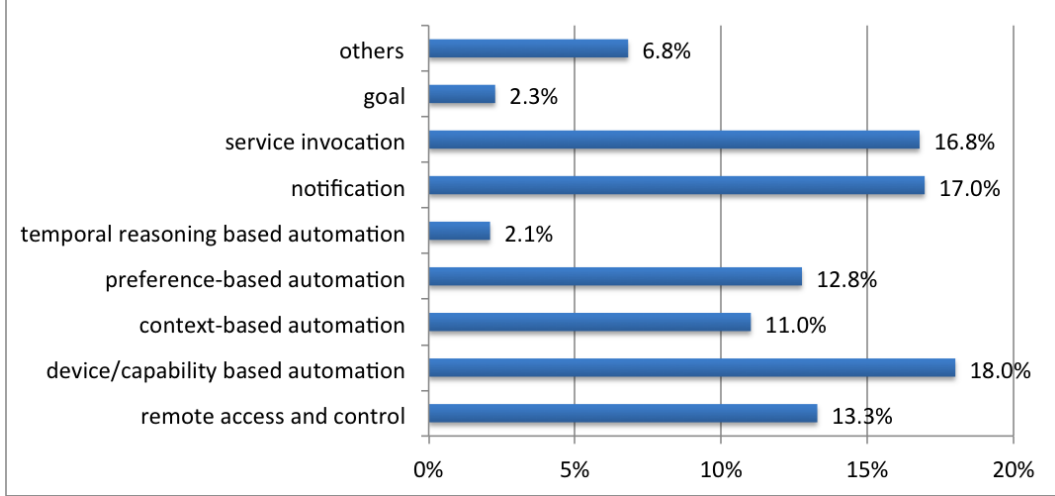


Figure 13: Distribution (%) of the nine feature categories from the user survey analysis

4.2.3 Categorization of SIoT Features

The responses from phase 2 and 3 are 572 smart home features written in English. We qualitatively analyzed the answers through two iterations. During the first iteration of our analysis, the possible feature categories were listed by examining all features collected to abstract them to high-level feature categories. We came up with nine different feature categories. During the second iteration of the analysis, the features from participants' answers were classified into the nine devised categories from the first iteration by labeling each answer with the most relevant category. The labeling was done by two researchers including the author and another Ph.D. student having knowledge of the *Socialite* system. If the categorization of an answer was not agreed upon by these two people, we discussed it until we reached a consensus category. The distribution of the nine categories are illustrated in Figure 13. The description and examples of each category are as follows:

- **remote access and control:** The features that end users want to monitor and control their connected devices using their smart phone or computer. Examples from the survey are “Being able to control the thermostat from multiple devices (desktop PC and smart phone, etc.) would be useful.” and “Start devices based on my remote input.”

- **device/capability-based automation:** The features having a combination of device capabilities to automate the smart home are classified into this category. Users mention either device types or device capabilities in their answers. The answers of “Automatically lock all doors and windows at the same time.” and “I would want the lights to come on when someone rings the doorbell.” are examples of this category.
- **context-based automation:** The definition of the context is often diffused and the perceived context can be different amongst users. The answers that require the combination of multiple devices and/or time and location to determine a condition are classified into this category. If a specified condition can be determined by different rules, an answer is classified into this example. For example, “Turn off TV when not watching.” is classified into this category, because determining *when not watching* can be achieved in various ways, for example by detecting user’s eye blinking or user’s movement. Another example for this category is “When I am going to sleep, turn off room lights”. Obviously determining a user’s context of *going to sleep* can be done by evaluating a set of device status, time and/or user’s location.
- **preference-based automation:** This category is for the answers that use a person’s preferred value for automation rules in a smart home system. An example from the survey is “If my friend’s smart phone is in my house, have the heater set to what she normally likes it based on the weather outside.” Another example is “Joe’s smart phone can ask his smart TV about shows that he likes to watch.”
- **temporal reasoning based automation:** Any answers that require temporal reasoning are classified into this category. Answers may include other automation categories discussed above, but we separate any examples with temporal reasoning because the realization of these features would require different approaches. Examples from the survey are “If the television detects that no one has used the remote *in a while*, send a message to the overhead light to turn itself off.” and “If the TV is on, and the motion detects no one is watching TV *after half an hour*, then turn off the TV.”
- **notification:** Answers that include notifying an alert or notifiable state change to the user’s smart phone or any displayable devices such as TV or computer screen. “If my smart phone is in my living room, and my TV is on, then voicemail played on screen.”

and “The oven in the kitchen finishes cooking and the message is displayed on my smart phone or TV if it is on.” are two examples from the survey.

- **service invocation:** Answers that require further processing of the information beyond the reasoning with the *Socialite* semantic model and/or accessing services not immediately available in the *Socialite* system are classified into this category. For example, ”If I am at home and it is around planned dinner, and nothing has been made, ask me if I’d like to order out.” and ”If TV is out of order, then ask other TVs in kinship who was used to repair TVs in past and satisfaction rating.” are related to this category. First example requires an access to an external service to make an order, and the second example requires a service implementation that can be done by analyzing the repair data in the *Socialite* history repository.
- **goal:** Answers that address a high level status of the devices belong to this category. It is expected that this feature category requires a user’s input, manufacturer defined value ranges or general consensus on the expectation with a numeric value to validate this feature. A participant responded that “I can check if my Brand X heating system is *working properly based on the usage data* from other Brand X heating systems around me.” In this example, *working properly based on the usage data* is ambiguous because it can be reasoned based on the device’s energy consumption ranking or the correctness of functions. However, the user’s interest is to know a high level status of their devices. This could be achieved with a set of other concrete rules.
- **others:** Answers that are either not directly related to the features and/or require other research and technology areas beyond this dissertation. For example, security and privacy concerns are important in SIoT but it is not the scope of this dissertation. Answers that are difficult to understand or are unclear to identify the intended meaning are also belong to this category. For example, answers with device types (e.g., TV) without any application scenarios/descriptions are classified into this category.

4.2.4 Observation of Relationship Types and Device Life-cycle Relevance

The features from co-locationship are mostly used together with friendship and thriendship. Users want to know preferences of their friends' devices when their friends are users' homes because it helps the user to be more sociable in the offline settings.

The most frequently referred feature for kinship includes diagnosis and repair related applications. For this kind of feature, the example scenarios often mention other relationships such as friendship and co-locationship, which implies that participants like to share with people who they know or can be identifiable because of their proximity. Participants show interests in sharing not only the repair and diagnosis related information but also proper settings and configurations for the devices in kinship. We observe that the features listed in the shared-ownership relationship address the automated sequence of the actions to achieve a high level task by sharing the device status (e.g., washer is done then start dryer).

With the above observation, the *Socialite* reasoning concept is designed to include the relationship types both in the condition and action expression in the rule description. Moreover, we allow the rules to be able to use union or conjunction of multiple relationships to select the devices that participate in each rule.

4.2.5 Acceptance of SIoT, End User Programming and Sharing Rules

Figure 14 show the distribution of programming experience, end user programming acceptance, sharing rules acceptance and SIoT acceptance from the participants.

In general, SIoT acceptance is lower than end user programming acceptance and acceptance of sharing of their rules with others. As for SIoT acceptance, the negative answer is low (20%) but the positive answer (28%) is also not high. 52% of participants answered "maybe" for SIoT acceptance. The reasons for the positive acceptance of SIoT concept include that the participants think our approach makes their life easier, secure, safe and save time. Furthermore, participants like the communication and interaction with devices because they can be informed about everything related to home when needed. Amongst the participants who are neutral (answered "maybe"), 48% of them concerned about security and privacy. Therefore, depending on the level of support for security and privacy they

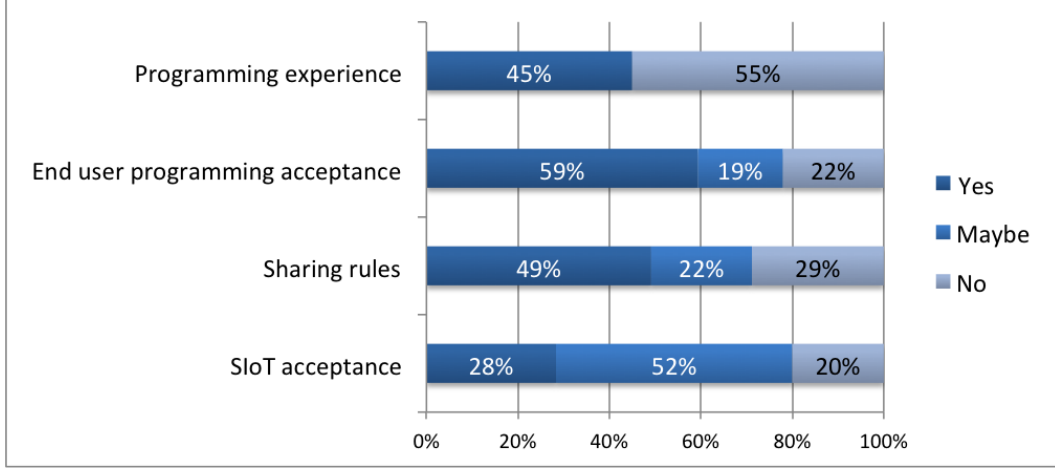


Figure 14: Distribution of programming experience, end user programming acceptance, sharing rules, SIoT acceptance

can be more positive in using systems in SIoT. Even among the participants with negative answers, 17% of them showed their concern with the security and privacy, while the rest of them are not interested at all in smart home in general.

In a previous study on Facebook privacy [19], the authors reported that privacy was a barrier for people who have not registered to Facebook, while privacy is less concerned for people who are already using Facebook. Also even for users who are using Facebook, their awareness of privacy control on Facebook was misconception.

Many researchers have identified an importance of the security and privacy for Internet of Things. In the SIoT paradigm, the security and privacy would be more concerned as shown in our study. Therefore a further research on what users' perceived privacy risks are and how their behavior would be adapted should be considered as future work.

The programming experience is not correlated to each of these factors: end user programming acceptance, SIoT acceptance and sharing rules in Pearson's chi-squared test results ($p > 0.05$). Also, either age or gender are not correlated with these factors in Pearson's chi-squared test results ($p > 0.05$). However, end user programming acceptance, SIoT acceptance and sharing rules are all correlated each other in Pearson's chi-squared test results

($p < 0.05$). End user programming acceptance is 59.32% and positive answers on sharing rules is 49.15%. This results implies that people are open to creating their own rules if an easy to use tool is provided.

4.3 NON-FUNCTIONAL REQUIREMENTS

This section discusses non-functional requirements for systems such as the *Socialite* system to realize the new paradigm of Social Internet of Things (SIoT).

Interoperability: Devices with Internet connectivity are available in the market. Typically, these devices are remotely accessible via the manufacturer’s mobile applications or Web pages that internally use private or public Application Programming Interfaces (API) defined by the manufacturer. Unfortunately these APIs are not standardized across domains and/or manufacturers, possibly because there is no strong business needs yet to make an effort for the standardization. Different manufacturers create their own applications to support different use cases. Therefore, an interoperable architectural solution is required to enable these connected devices to interact with each other.

Scalability: We expect that the communication of devices, users and *Socialite* system will become larger in SIoT than other applications of Internet of Things because more interactions are necessary to share and discover new information through explicit relationships and reasoning. In case the adoption rate of the new paradigm of SIoT will increase over time, the architecture should be able to support horizontal scalability, which means that the architecture should support to add more nodes to a system: add a new computer to a distributed software system. At the same time, the real-time processing of the data at scale [46, 28, 123] is one of the drivers in the development of the software framework.

Extensibility: A new device from a different manufacturer, which is not foreseen during the development time should be supported, because a new company may produce devices used by many people in Internet of Things. A new type of device can also be introduced in addition to the devices we already know. The architecture should be extensible to support new device types. To meet the different needs from various people, the system should be

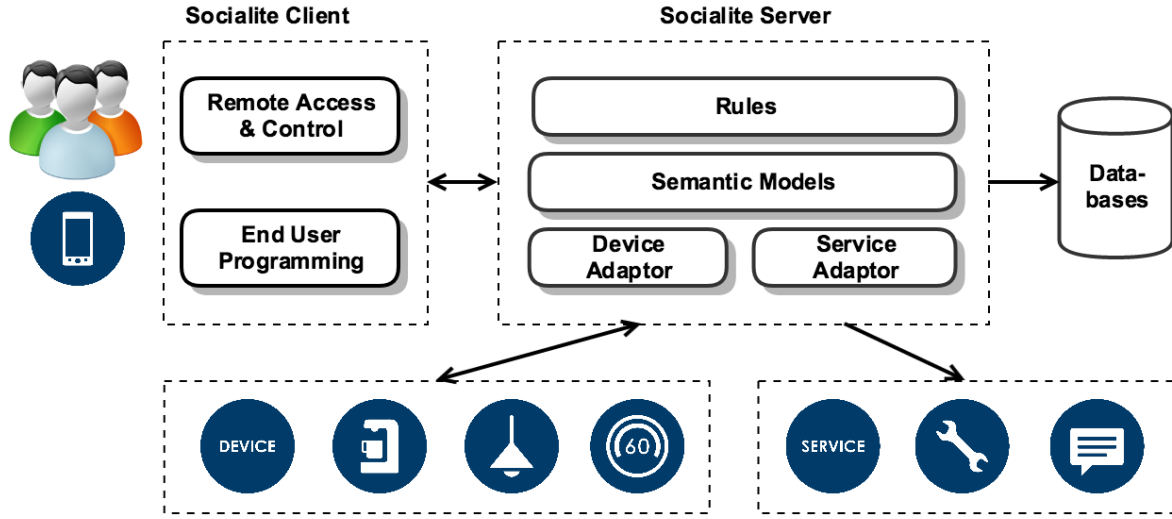


Figure 15: *Socialite* system overview

extensible in terms of the future applications the users (both end users and developers) also would come up with.

4.4 SOCIALITE SYSTEM OVERVIEW

Figure 15 graphically represents an overview of the *Socialite* System, which consists of Web based *Socialite Client* application, a distributed *Socialite Server* accessing devices and services and various *Databases*.

The users of the *Socialite* can *remotely access and control* their connected devices using the *Socialite Client* application through a web browser on their computers, tablets or smart phones. The core functions of the client application include the management of users' devices and relationships, and remote access and control of connected devices in relationships. In addition, users can create their own rules (e.g., if my friend is in my living room, set the thermostat temperature to my friend's preference value.) through *end user programming*,

which is an enabler of the *extensibility* requirement.

The *Socialite Server* provides a uniform access to heterogeneous devices from different manufacturers by decoupling common data models (represented in *Semantic Model*) and manufacturer specific device implementations in *Device Adaptor*.

The *Semantic Model* is used to describe *Rules* defined by end users as well as system administrators, which run on the reasoning engine to make an automated decision upon any events (e.g., device status change, user's location change).

Similarly, any REST services provided by third parties are uniformly accessible through *Semantic Model*, which is decoupled from the implementation of *Service Adaptor*. The semantic model and related architecture are our approach to enable *interoperability* and *extensibility*.

The communication between the *Socialite Client* and *Socialite Server* are two kinds: *HTTP REST services* are used for management of user profile, devices, relationships and rules, and *Web socket* is used to get a notification on changes for the subscribed event to the *Socialite Server*. The communication protocols between *Devices* and *Socialite Server* are not restricted by our architecture and design. However, the current practice of device APIs from the manufacturers offers *HTTP(S) REST services* with JSON payload. The third party service invocation is only through *HTTP(S) REST service call*. The *Socialite Server* uses remote procedure calls to access the various *Database systems*, which manage persistent data (e.g., semantic models, device history and rules).

The *Socialite Server* is an event-driven architecture where asynchronous communication is intrinsic communication mechanism and a basis for our solution to support *scalability*. The reasoning engine that runs *Rules* performs computationally intensive tasks for the evaluate of relevant rules in the engine upon a new event (e.g., device status change). Therefore, reasoning engines are distributed over multiple nodes to evaluates rules for different users in parallel and to accomplish *scalability* by integrating an open source data stream processing solution developed by a social network platform (Twitter).

4.5 SOCIALITE SEMANTIC MODELS

The devices, users and their relationships together with other information are represented using the formal languages used in the Semantic Web, often used for data integration and interoperability [131, 42]. This section first provides the basic knowledge about ontology and technologies used in Semantic Web. The details of the *Socialite* semantic model is then introduced.

4.5.1 Background of Technologies in Semantic Web

An *ontology* is a specification of a conceptualization in the context of knowledge sharing [81]. It refers to the kinds of *objects* that will be important to the agent and the *properties* those objects will be thought to have, and the *relationships* among them [49].

The Semantic Web [41] is a set of standards and best practices for sharing data and the semantics of that data over the web for use by applications. A set of standards include the Resource Description Framework (RDF) [103, 28] data model, the SPARQL query language [140], the RDF Schema (RDFS) [50] and Web Ontology Language (OWL) [119] standards for storing vocabularies and ontologies [72]. The Semantic Web is used for data integration [21].

The RDF is a language for the representation of resources, which can be anything that someone might want to talk about [21]. The *triple* is the fundamental data structure of RDF. A triple is made up of a *subject*, *predicate*, and *object*. A set of such triples is called an RDF graph. An RDF graph can be visualized as a node and directed-arc diagram, in which each triple is represented as a node-arc-node link (see Figure 16). Triples are the statement about resources, using Uniform Resource Identifiers (URIs), and literal values such as strings and integers.

The serialization (the process of converting an object into a stream of bytes in order to store the object or transmit it to memory, a database, or a file) format of RDF data include Turtle [37], N-Triples [36], N3 [40], RDF/XML [35] and JSON-LD [158]. A triple store, also known as an RDF store, is a database that is tuned for storing and retrieving data in the

form of triples. In addition to the familiar functions of any database, an RDF store has the additional ability to merge information from multiple data sources, as defined by the RDF standard. RDFS and OWL are World Wide Web Consortium (W3C) standard vocabularies that supports to define and describe classes and properties for the user’s dataset and allow a certain application infer new information from the dataset.

The SPARQL is the W3C standard query language for RDF query language. The triple store often provides a SPARQL endpoint, which accepts SPARQL queries delivered over the Web and returning results in a choice of serialization formats.

4.5.2 Socialite Semantic Model Description

Semantic models in the *Socialite* system represent user and device information together with its attempted repair solution, static and dynamic locations of users and devices, services accessible through REST interfaces and relationships between users and devices on the *Social Internet of Things*.

Our semantic models are formally described through RDF language [103, 28], which are knowledge representation languages for authoring ontologies as discussed in Section 4.5.1. The ontologies translate the domain of interest into a set of triples.

The goal of the *Socialite* ontology is to have a minimum set of models, which can address the interoperability of different devices, use the device model for various device life-cycle phases including operation and diagnosis/repair phases. The user model aims to represent a user’s profile required for interaction with devices and other users through new social relationship types proposed in Section 4.1.1. The location model is developed to represent the user and device’s location often used in the context-based automation, and represent

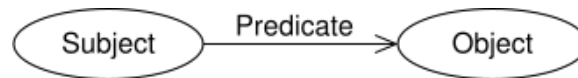


Figure 16: RDF graph with two nodes (Subject and Object) and a triple connecting them (Predicate)

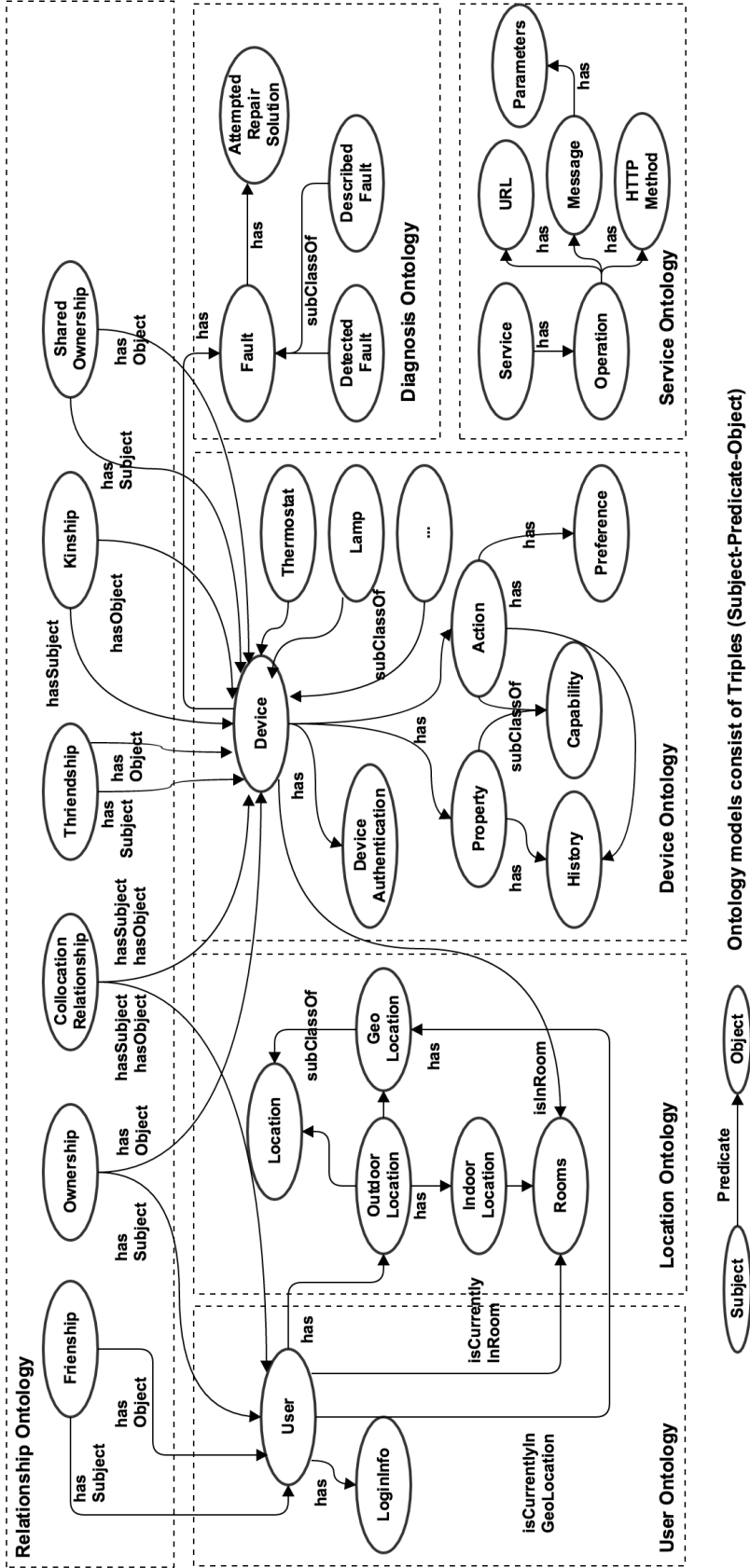


Figure 17: Graphical representation of the core ontologies in *Socialite* (Shortend)

co-location relationship. The service model is introduced in our semantic model to support the service invocation feature from the user survey.

Figure 17 graphically illustrates the representation of ontology models for *User*, *Device*, *Location*, *Relationships*, *Device Diagnosis*, and *Service* used in the *Socialite* framework. The *User* ontology is associated with the *Device* ontology through the *Relationship* ontology. The *User* ontology has a link to the *Location* ontology having their home address as well as their current *Locations*.

The *User* requires *LoginInformation* to access a *Socialite* runtime instance. The *Socialite*'s user model can be later extended from the existing ontologies such as friend-of-friend (FOAF) [51] if the *Socialite* system needs to integrate existing social systems in the future. However, the adoption of FOAF is low and it does not support different relationships that the *Socialite* proposes. Therefore, we develop our own *User* model to support the *Socialite* use cases and features discussed earlier in the chapter.

The *Socialite*'s *Location* ontology is used to specify the location of both *User* and *Device*. A *User* has *OutdoorLocation*, which is a *subClass* of *Location*. *OutdoorLocation* has *GeoLocation* and one or more *IndoorLocation*, each of which has *Rooms* associated with it. The *Location* ontology does not model different rooms, which is the case for the other smart home ontology such as DogOnt[45]. We rather allow users to define their own room types for their home.

The *Device* ontology is carefully developed to support interoperability of the heterogeneous devices. The *Device* model addresses semantic interoperability coming from different data models used in manufacturers' APIs and payload. It is a light-weight model compare to the device model used in our smart home gateway (see Chapter 3) that is extended from DogOnt. For example, the network and communication model to enable device and service discovery is not considered in the *Socialite* ontology. The reason is the fact that the heterogeneous communication protocols are not required in the *Socialite* system and the inferred knowledge from the hierarchy of the functionalities is also not required for the *Socialite* use cases.

Device has various *Capabilities*. Specific device types, such as *Thermostat* and *Lamp*, are *subClasses* of *Device*. *Properties* and *Actions* extend *Capability*. *Properties* represent device

status (e.g., get current temperature), while *Actions* represent control/actuation commands (e.g., set the target temperature to a certain value). *Property* contains data *History*, while *Action* contains both *Preference* and *History*. *Preference* for *Action* value (e.g., the preference for the target temperature set-point is 75°F) can be either given by the user’s input about his/her target goal, or inferred by a set of rules or algorithms. The *Property* value comes from usually the environment sensing (e.g., current temperature is 75°F), thus it does not require any *Preference*. *Actions* and *Properties* provide a generic representation of the capabilities provided by each device type regardless of the manufacturer.

In order to handle failures possibly occurring during the life-cycle of devices, a *Diagnosis* ontology is introduced. *Device* has *Fault*, which can be either *Detected Fault* such as an error code from a device, or *Described Fault*, for example a symptom that a user of the device can describe (e.g., the hot water is not very hot). Each *Fault* has the list of *Attempted Repair Solutions* together with the result (either success or not) of the attempted action.

The *Service* ontology uses the RDF/OWL description of the hRESTS model [105]. As discussed in Section 2.2, hRESTS is an HTML microformat for describing RESTful Web Services. The microformat [3] is a semantic markup using HTML/XHTML tags for meta-data, and other attributes in web pages. The hRESTS describes main aspects of services, such as operations, inputs and outputs. The *Service* ontology is a RDF representation of hREST model.

The *Socialite Relationships* (See in Table 2 in Section 4.1.1) are associated with *User* and *Device*. *Friendship* is only allowed between users, while *Kinship*, *Shared Ownership* and *Thriendship* are only allowed between *Devices*. *Co-location* is allowed for both *Users* and *Devices*.

4.6 SOCIALITE SERVER ARCHITECTURE

The *Socialite* server architecture is an event-driven architecture that enables large scale data stream processing with distributed reasoning engines. The *Socialite* semantic model discussed in Section 4.5.2 is integrated to the system as a solution for *interoperability*. The

decoupling of the objects that represents semantic models and the detail implementation of how to access the device’s application programming interfaces and the service interfaces is an enabler to achieve *interoperability* and *extensibility*. The system is designed also to support *scalability* by using an asynchronous communication for various software modules, and also by distributing computationally intensive processing tasks (reasoning engines) over multiple nodes and by employing a NoSQL database for the device history.

4.6.1 Interoperability for Various Manufacturer’s APIs

As discussed in Section 4.5, the *Socialite* device model abstracts common attributes for all device types and all the way down to a specific device type. The semantic model that conceptualizes the common attributes for a typical device type is able to unify implementation variations from different manufacturers. The *Socialite* architecture decouples the implementation details of how to access devices from its common semantic model. This section discusses how the similar types of devices from different manufacturers are modeled and managed in the system.

4.6.1.1 Physical and Logical Devices We observed that the different manufacturers use different physical components for the same type of devices. For example (see Figure 18), a thermostat from manufacturer A may have capabilities for measuring the current temperature, and also for detecting occupancy. Another thermostat from manufacturer B may be only capable to measure the current temperature.

Socialite aims to support interoperability of devices from different manufacturers. Therefore, we define a common set of capabilities of each device type, and allow manufacturer specific capabilities to be mapped to other types of devices. The real devices are called *physical devices* and devices represented in the *Socialite*’s semantic model are called *logical devices*. In Figure 18, the *Thermostat from Company A* is represented as two *Logical Devices*, which are (a) *Motion Detector with Detect Occupancy Capability* and (b) *Thermostat with Measure Temperature Capability*. *Socialite* is designed to share the common information, such as the device authentication, between the different *logical devices* from the same *physical device*.

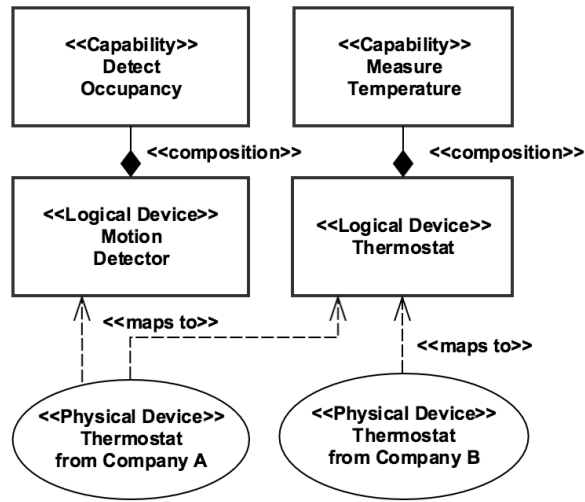


Figure 18: Example of physical and logical devices of thermostat

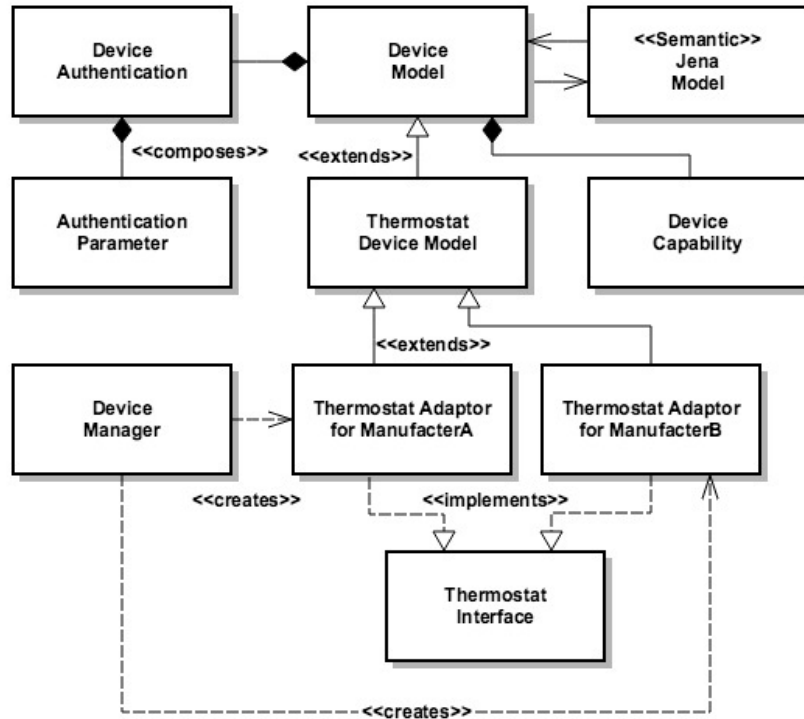


Figure 19: Common semantic models for devices

4.6.1.2 Common Device Model and Manufacturer Specific Device Implementation As shown in Figure 19, the *Device Model* consists of *Device Authentication* and *Device Capabilities*. *Device Authentication* is always required to access the registered device remotely; however different manufacturers of the same type of devices often use different authentication mechanisms.

For example, a thermostat from *Manufacturer A* uses OAuth 2.0 [89] for the device authentication while another *Manufacturer B* requires providing the device serial number and “the secret” (such as an authentication token). Also, the APIs and the payload are very different across manufacturers. In our architecture, *Device Authentication* consists of any number of *Authentication Parameters*, which are key-value pairs.

When the *Device Manager* creates an instance of the *Thermostat Device Model*, it can create a proper *Thermostat Adaptor* implementing the device authentication for that thermostat. By doing this, we can keep the consistency for the common attributes of the *Thermostat Device* type while allowing different authentication mechanisms to be used for different devices.

4.6.2 Persistent Management and Repositories

The *Socialite* server manages to store three different types of information in the database systems. The *Socialite* semantic models (see Section 4.5) and its instances are stored in the Triple store (RDF store), the history of device actions and properties, and user defined rules (see Section 4.7.1) are stored in separate instances of the NoSQL databases. This section explains how the *Socialite* manages to add, update, delete and retrieve the information stored in these three different repositories.

4.6.2.1 Semantic Model Management and Its Repository Although Semantic Web technologies have been available for decades, their complexity has hindered the emergence of real implementations of Semantic Web services [38]. The *Socialite* server aims to reduce this barrier during the software development by using Java object classes with RDF/OWL annotation that is supported by JenaBean [61].

For example, in Code 2, *Relationship object* is a plain Java object with nothing but Java annotations for the RDF/OWL attributes. A Java object with JenaBean annotations is mapped to an object in the form of Resource Description Framework (RDF) model used in the Jena framework [91], which is an open source Semantic Web framework for Java. The Jena framework provides an API to read and write to RDF models. Code 3 provides a Turtle [162] representation of an excerpt of the RDF model corresponding to JenaBean in Code 2.

Code 2 Example of Java class with JenaBean annotation

```

1 import thewebsemantic.*;
2
3 @Namespace("http://siot.pitt.edu/ontology#")
4 public class Relationship implements Serializable {
5
6     //This is an excerpt of Relationship Class.
7     //Not all variables and methods are shown here.
8     protected String relationshipId;
9     protected S
10
11     @Id
12     @RdfProperty("http://www.w3.org/2000/01/rdf-schema#" + "label")
13     public String getRelationshipId() {
14         return this.relationshipId;
15     }
16
17     @RdfProperty("http://siot.pitt.edu/ontology#" + "hasObject")
18     public SIoThing getObjectThing() {
19         return objectThing;
20     }
21
22     @RdfProperty("http://siot.pitt.edu/ontology#" + "hasSubject")
23     public SIoThing getSubjectThing() {
24         return subjectThing;
25     }
26     //End of an excerpt of code
27 }

```

The *Socialite* semantic models represented in the form of RDF are stored in the *Triple Store*. We use the Virtuoso database [75] to store RDF triples. RDF triples are inserted, updated, retrieved and deleted through a semantic query language, namely SPARQL [140]. Our software provides methods that abstract the SPARQL query details. SPARQL query results (e.g., all devices located in a user A's kitchen) are retrieved as *Jena Model* objects, which then are mapped to *JenaBean* objects. By doing so, developers without in-depth

Code 3 Example of resource representation in Turtle format

```
1 @prefix siot: <http://siot.pitt.edu/ontology#> .
2 @prefix owl: <http://www.w3.org/2002/07/owl#> .
3 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
4 @prefix xml: <http://www.w3.org/XML/1998/namespace> .
5 @prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
6 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
7 @base <http://siot.pitt.edu/ontology#> .
8
9 <http://siot.pitt.edu/ontology#> rdf:type owl:Ontology .
10
11 siot:Relationship rdf:type owl:Class .
12
13 siot:hasSubject rdf:type owl:ObjectProperty ;
14     rdfs:domain siot:Relationship ;
15     rdfs:range siot:SIOThing .
16
17 siot:hasObject rdf:type owl:ObjectProperty ;
18     rdfs:domain siot:SIOThing ;
19     rdfs:range siot:SIOThing .
20
21 siot:relationshipId rdf:type owl:DatatypeProperty ;
22     rdfs:domain siot:Relationship .
```

understanding of the Semantic Web technologies can easily work on the *Socialite* system because they only deal with Java objects rather than RDF models.

4.6.2.2 Device History Management and Its Repository We expect that device history will grow, requiring horizontal scaling. Horizontal scaling means to add more nodes to a system: add a new computer to a distributed software system. Because horizontal scalability is not mature in triple stores, we employ NoSQL database systems that are designed to support horizontal scalability better than relational database systems and RDF database systems [88]. The *Socialite* architecture uses MongoDB [120] to store the device history while managing the device semantic model in the *Triple Store*.

Consider how the device history is updated in the *Socialite* framework. There exist two types of manufacturers' APIs with respect to the device status change: 1) *Push*: the manufacturer's server pushes the change to *Socialite*, which subscribes to the changes of the registered devices and 2) *Poll*: *Socialite* needs to call the APIs periodically to keep track of

the changes.

Once the *Socialite* gets informed about the changes either in *push mechanism* or *poll mechanism*, changes are placed in the *Device Message Queue* where the *Device Listener* consumes the message to update the device history in the MongoDB database and replace the previous value with a new changed value via *Device Manager* and *Persistent Manager*. The changes are propagated all the way down to the *Persistent Manager*, which is in turn responsible for updating the history repository and semantic model repository.

4.6.2.3 Rule Management and Its Repository The *end user programming* features in *Client Application* allow end users to define their own rules and share them with other users through the *Socialite framework*. The rules in the *Socialite* server are internally represented in a domain specific language with the rule syntax defined by the production rule system used in the system. However, we do not envision that client application developers are familiar with the rule syntax of Drools [18]. Rather, the *Socialite* server exposes rule-related APIs in a plain JSON format, which can be translated into a proper rule syntax by providing a rule translator.

The *Rule manager* in *Socialite* adds more attributes in addition to the attributes required for the Drools reasoning engine, because those are not sufficient for our purpose of rule management and sharing. The rule management supports activating and deactivating rules as well as adding, editing and deleting rules. As discussed in Section 4.7.1, the rule has a *sharing* attribute that can take the values of private, public and constrained by relationship. These attributes are supported in the Rule management APIs.

When a new rule is added to the system through the *REST* interface, a rule message is created and placed in a rule queue. The subscribers of the rule message include the rule persistent manager, which stores the JSON payload in the MongoDB database. The reasoning engine, which is another subscriber to rule messages, translates the rule in JSON payload to Drools syntax and puts the translated rule syntax into the reasoning engine. Similarly, rule deletion messages are subscribed to by the reasoning engine as well as *Persistent Manager*.

During the initialization of the *Socialite* server and reasoning engines, all rules read from the rule repository are inserted into the reasoning engine.

4.6.3 Scalability with Event-Driven Architectural Pattern

The goal of the *Socialite* event-driven architecture is to handle any events from devices and users, as well as the results from their interactions with the system in a distributed and asynchronous manner. The events produced by devices and users are dispatched to the listener (with specific periods and other temporal characteristics) that subscribes to the events for various purposes including updating status of devices and users, executing rules, visualizing the information and managing persistent repositories. The distributed system architecture across the multiple reasoning engines support processing large scale events through multiple channels. The intelligent rules specified as complex events are to be executed within the time constraints specified for each application.

This section provides background information of event driven architecture and Java Message Service. It discusses how we model and design events and related architectural components for the *Socialite* system including event channels, event generators, event processing styles.

4.6.3.1 Background of Event-Driven Architectures An event-driven architecture is a distributed service-oriented architecture in which all communications are through events and all services are independent, concurrent, reactive event driven processing (i.e., they react to input events and produce output events) [111]. Therefore, it provides a loose coupling of its components. A component publishes events and other components can subscribe to these events. The producers/publishers and consumers/subscribers are completely decoupled from each other. The event is distributed across the event processor components through a lightweight message broker such as ActiveMQ [155] with which the *Socialite* framework integrates. There are two main types of architectural components within the message broker: a broker component and an event processor component. The broker component can be centralized or federated and contains all of the event channels that are used within the event flow. The event channels can be message queues, message topics, or a combination of both [121]. Figure 20 illustrates a broker topology of the event-driven architecture. All events in the *Socialite* system are created as a form of message.

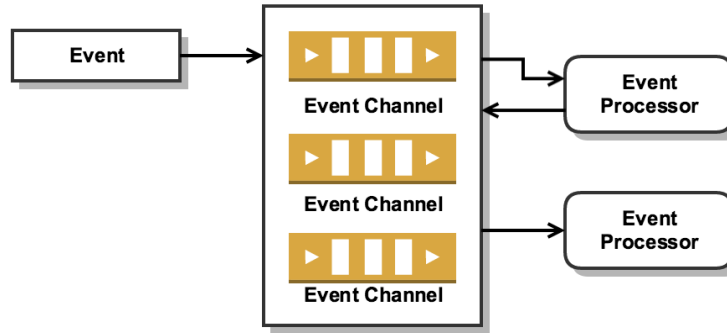


Figure 20: Broker topology in event-driven architecture

Java Message Service (JMS) [143] is a specification that describes a common interface to standard messaging protocols and also to special messaging services in support of Java programs. It allows for communication between different components of a distributed application to be loosely coupled, reliable and asynchronous. Therefore, JMS is regarded as a solution for reducing the system bottlenecks and increasing overall system scalability [143]. When a message is sent, it is addressed to a destination (i.e., queue or topic), not a specific application. An application that subscribes to or registers an interest in that destination may receive the message. In this way, the applications that receive messages and those that send messages are decoupled. Senders and receivers are not bound to each other in any way and may send and receive messages as they see fit.

ActiveMQ [155] is an open source message broker that implements the JMS specification. It provides persistence and, once and only once assurance for message delivery. The message broker is responsible for creating and managing network connections and messages used for communication. It supports the producer-consumer message model as well as the publisher-subscriber model.

4.6.3.2 Socialite Event Channels An event represents a change of state. It usually consists of a header and a body. The header contains meta information such as its name,

time of occurrence, duration, and so on. The body describes what happened. For example, if an event is generated due to a device status change, the event header contains an event type and a timestamp of the status change, and the event body includes the device object with the capability name and value for an update.

The *Socialite* system has the following event channels/queues, which cover all possible events in the *Socialite* system:

- **Device channel:** Events related to the device life-cycle management are placed in this channel. The event types used in this channel are a new device registration, device status change, and de-registration of a device from the system. The diagnostic/fault information from the device is also categorized as the *Device* event because the fault is a part of the device status change.
- **User channel:** Events related to the user management are placed in this channel, including a new user registration, a user profile update and de-registration of a user from the system.
- **Location channel:** Events related to the location management such as a creation of a new location for a user or device, or update of a user's current location use this channel.
- **Relationship channel:** This channel is used for events related to the relationship management such as a new relationship generation and removal of an existing relationship.
- **Rule channel:** Rule life-cycle related events are placed in this channel, which are a new rule creation for a user, editing a rule, activating or deactivating a rule, removing a rule from the system.

4.6.3.3 Socialite Event Generators Events in *Socialite* are generated from variable sources including sensors, REST API calls and software components inside the *Socialite* system. Different events are formatted into a uniform message format prior to being placed in the event channel. The occurrence of these events can trigger the invocation of one or many services through either producer-consumer or publish-subscribe architectural pattern. The producer or publisher of an event puts an event in a designated queue. The message includes a message type corresponding to each event type's fine-grained classification (e.g., update device status) so that the interested parties can handle a message efficiently. The

message body includes a different object (e.g., *Device object*, *Rule object*) upon the message type. The consumer/subscriber processes a message according to the appropriate event processing logic. The three event processing styles are addressed in Section 4.6.3.4.

The current states of users devices, locations and relationships can be updated through the client application, connected devices or actions from the reasoning engine. Figure 21 illustrates how the events generated from the devices, the client application and the reasoning engine are processed differently throughout the *Socialite* system; these three ways of generating events are illustrated by examples.

Example 1: Device status is changed from the connected physical device

Consider a thermostat and a humidity sensor installed at a user’s home. If a device status is changed either by a user through physical interaction with a device (e.g., changing a target temperature through a physical device) or by another sensor through its physical sensing capability (e.g., humidity sensor value is changed), the corresponding *Device Adaptor* is updated. Some manufacturers detect the change and the manufacturer’s server publishes the change to *Socialite’s Device Adaptor*, which subscribes to the change. If this option is not implemented by the manufacturer, *Socialite* implements a task that regularly polls the device status and updates the *Device Adaptor* if any difference is detected. When that happens, the *Device Adaptor* calls the *Semantic Model Manager* to update the device status. The *Semantic Model Manager* enhances the change information with semantic information of the device and then notifies the *Event Broker* about the change. The subscribers evaluate and process the event. For example, the *Persistent Manager* subscribes to the device status change event and updates the records in *Triple Store* for the current status and *MongoDB* for the history. The *Pub/Sub API Service* subscriber broadcasts it to the *Client Application*. The *Reasoning Engine* subscriber inserts the updated device object to fire rules that meet the condition triggered by the device change.

Example 2: User’s location is updated from Socialite REST API invocation in the client application Consider that *Client Application* (either through GPS information in smart phones, or manual check-in of a location on UI) informs to the *Socialite* server of the change of the user’s current location. The REST service calls all functions to update the *Semantic Model* with the change. Then *Semantic Model* notifies the *Event Broker* with the

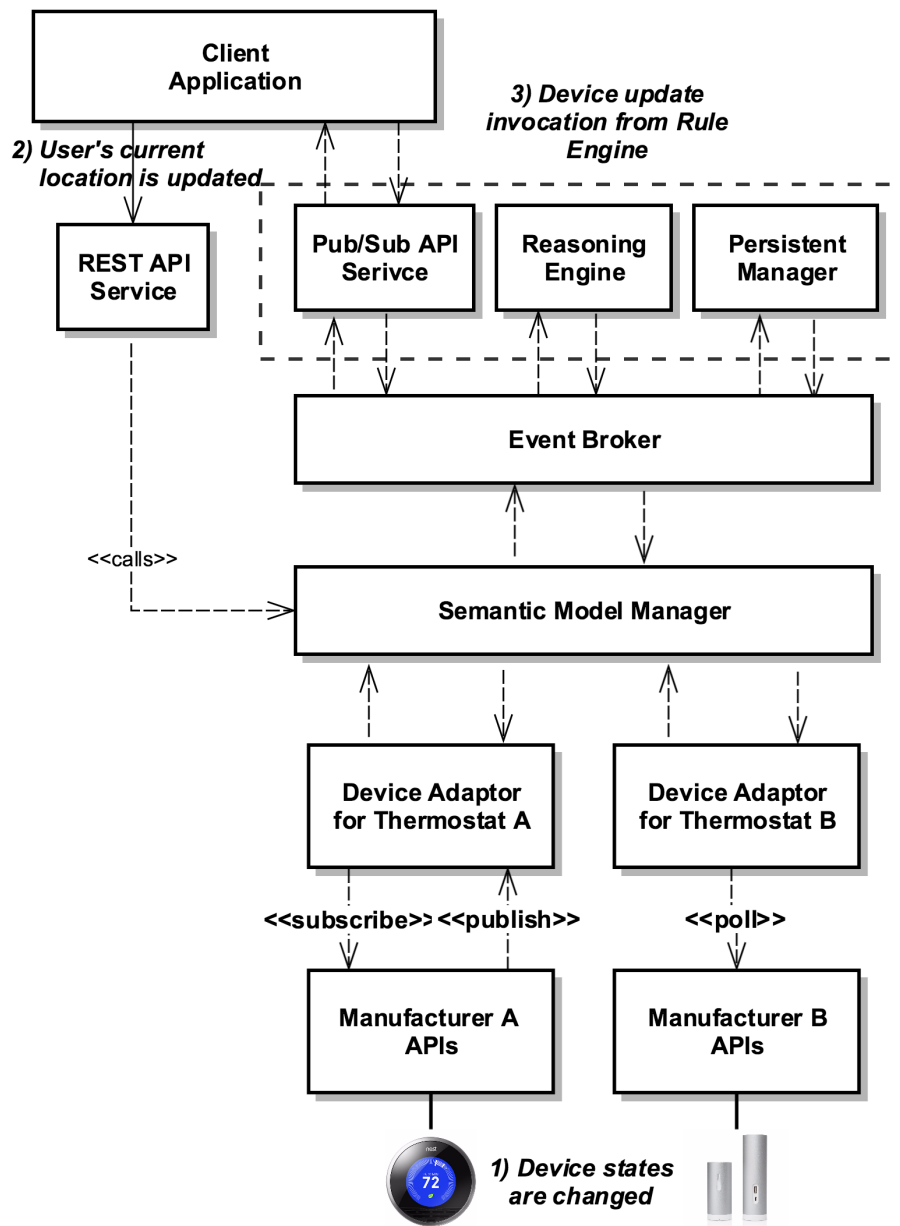


Figure 21: Status update from (1) the physical devices, (2) the client application, and (3) from the reasoning engine

change. The *Reasoning Engine* is notified with this change and triggers relevant rules, such as updating the relationships (e.g., co-location relationship). The updated relationships are again published from *Event Broker*. The rules matching the condition of this new relationship change will react upon the updated relationships.

Example 3: Device status update invocation is called from the Socialite Reasoning Engine The *Socialite Reasoning Framework* discussed in Section 4.7.1 allows a user to specify rules that execute various actions upon a triggered event. For example, a device status change event described in *Example 1* and a user’s location change described in *Example 2* trigger actions. These actions include the internal service invocation in the *Reasoning Engine* to generate new *Device status change event* that are eventually notified to the *Semantic Model Manager* through *Event Broker*. The *Semantic Model Manager* is responsible for updating the device semantic model and calling the *Device Adaptor* to request for changing the real device value by accessing the manufacturer’s APIs.

4.6.3.4 Event Processing Styles There are three general styles of event processing in the event driven architecture, which are *simple*, *stream* and *complex* event processing [121]. This section addresses how these three styles of the event processing are used in order to accomplish various functions and reasoning required for the *Socialite* server.

- **Simple event processing:** Simple event processing includes processing such as changing the events schema from one form to another, augmenting the event payload with additional data, redirecting the event from one channel to another, and generating multiple events based on the payload of a single event [125]. Examples of simple event processing in *Socialite* include a new device is added, or a new relationship is created. The processing logic assesses event type and content and then reacts accordingly. The simple event processing style is used for the persistent manager to store the history in the *Socialite* system upon event occurrence.
- **Stream event processing:** Stream event processing is commonly used to drive the real-time flow of information in and around the system enabling real-time decision making [121]. Device status change events either pushed by devices or polled by the server are filtered for a notability (e.g., temperature value is changed from the previous value by a

certain threshold) and streamed to subscribers (e.g., persistent manager to update the history) in the *Socialite* system. We expect that a great portion of the rules created by the users are related to this type of event processing.

- **Complex event processing:** Complex event processing is primarily an event processing concept that deals with the task of processing multiple events with the goal of identifying the meaningful events [110]. Patterns spanning multiple independent events are detected in order to derive new complex events, which are events that summarize, represent or denote a set of other events. Typically, this processing involves applying a collection of evaluation conditions or constraints over an event set. The events may span different event types and may occur over a specified time period [121]. Complex event examples are used to evaluate the sequence of the events (e.g., *event 1* happens before *event 2*), if a number of events happened in a specific time frame. Examples include a rule to evaluate if the temperature of the sensor values is changed more than 10 ° F over last 10 minutes, or another rule to evaluate if the lamp on-off state has been toggled more than 10 times over last one minute. The usage of the complex event processing in the *Socialite* reasoning engine is discussed in Section 4.7.1.2.

4.6.4 Large Scale Reasoning over Data Streams

The *Socialite* server architecture is designed to be scalable to handle large events/messages to make automated decisions based on the rules created by the end users. As a solution for scalability, the *Socialite Server* architecture processes the events that are notified to the *Reasoning Engine* in a distributed manner. The *Socialite* Reasoning Framework (see Section 4.7.1) distributes the subscribed events to multiple data stream channels so that a parallel and asynchronous processing of rule evaluation is scalable and fast compared to the architecture with a single reasoning engine.

This dissertation studies our solution in existing social network systems, which are in fact advanced in terms of the handling distributed, real-time and large-scale data streaming. The *Socialite* system leverages Apache Storm [159], which is an open source software used to run various critical computations in Twitter at scale, and in real-time [161].

The remaining of this section provides background information of Apache Storm and the *Socialite* topology for distributed reasoning.

4.6.4.1 Background of Apache Storm Many modern data processing environments require processing complex computation on streaming data in real-time [161]. This is particularly true in Social Internet of Things systems including the *Socialite* system, where each interaction with users and devices often requires making a number of complex decisions, based on data that has just been created.

Storm is designed to be *scalable* to support adding or removing nodes from the *Storm cluster* without disrupting existing data flow through *Storm topologies*. It supports fault-tolerance to provide fail over solutions against hardware component failures in a large cluster. *Storm* offers good performance characteristics required in real-time applications [161].

The *Storm cluster* is made up of a main node and several working nodes as shown in Figure 22. A daemon process called *Nimbus* is running on the main node, in order to allocate codes, arrange tasks and detect errors. Each working node has a daemon process called *Supervisor* to monitor, start and stop working processes. The coordination work between *Nimbus* and *Supervisor* is handled by *Zookeeper*. *Zookeeper* is responsible for coordinating the various nodes (e.g., *Nimbus*, *Supervisor* daemons) within a *Storm cluster*.

In order to realize real-time computation on *Storm*, topologies should be created inside of it. *Topology* is a direct graph of processing logic nodes called *Spouts* and *Bolts* generating data streams in the form of *Tuple*. Typically *Spouts* pull data from queues such as the channels/queues in the *Socialite* system. The way *Bolts* and *Spouts* are connected indicates how data should be passed around between these nodes, which makes up the topology. *Bolts* process the incoming *Tuples* and pass them to the next set of *Bolts* downstream if needed.

Storm topology provides various grouping to route the *tuple*. The following groupings are used in the *Socialite* system:

- *fields grouping*: The stream is partitioned by the fields specified in the grouping. For example, if the stream is grouped by the *username* field, tuples with the same *username* will always go to the same *bolt*, but tuples with different *usernames* may go to different *bolts*.

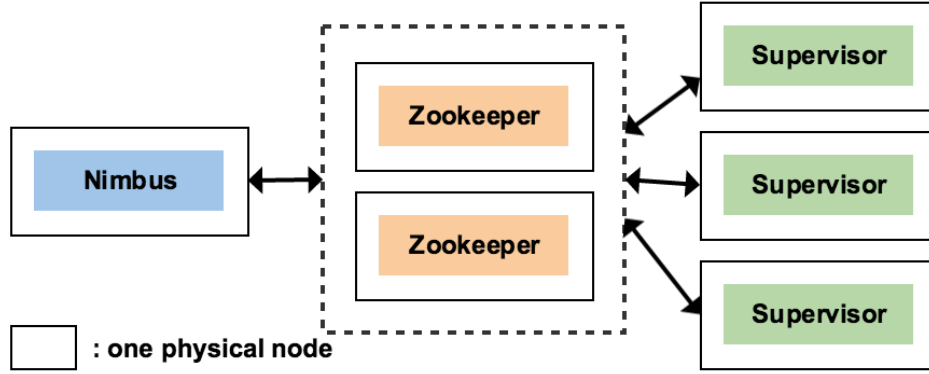


Figure 22: Storm cluster concept

- *all grouping*: The stream is replicated across all the *bolts*.

In addition to *fields grouping* and *all grouping*, *Storm* provides five more built-in stream groupings, which are *shuffle grouping*, *partial key grouping*, *global grouping*, *none grouping*, and *direct grouping*. The following section explains why we choose *fields grouping* and *all grouping* and how we use them in the *Socialite* system.

4.6.4.2 Socialite Storm Topology The *Socialite* server aims to support the large number of users expected in the Internet of Things. The *Storm* architecture is used to scale the system horizontally. The bottleneck of the *Socialite* system is the reasoning engine because it is computationally intensive: to evaluate all events disseminated from various sources and types of events (user, device, relationship, location and rule) and to fire the rules if evaluation is true.

One challenge with that is how the system distributes the events evenly by preserving consistent destination for all device objects of which previously inserted status are used for the evaluation of the rule. In particular, if a device owned by one user is in one processing *Bolt*, and another device owned by the same user may be processed in a different *Bolt*. The triggering conditions in the *Socialite* rules are expected to be mostly based on the devices

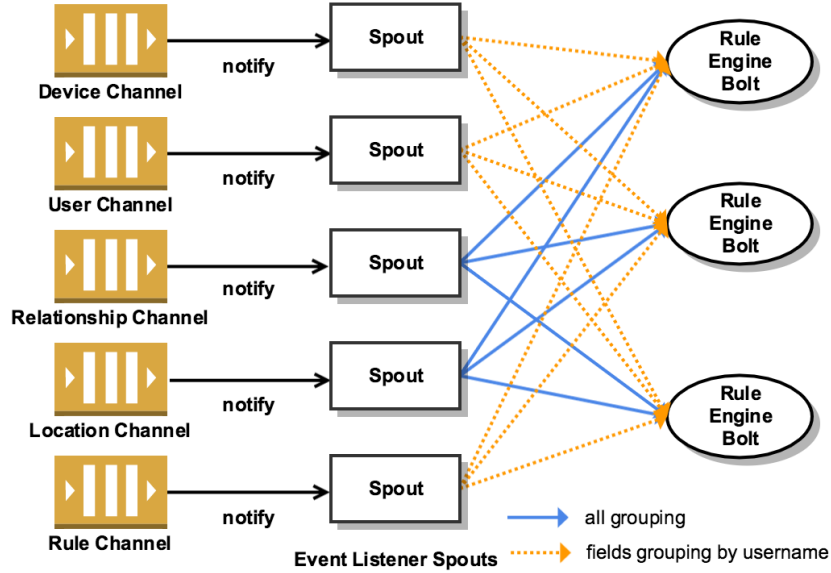


Figure 23: Storm topology in *Socialite*

with ownership relationships while the actuation can be any devices in all relationships and services. Therefore, the parallel event processing is done based on the user identification by configuring the reasoning engine *Bolts* with *fields grouping* of *username*, which is unique to an individual user. The *fields grouping* sorts the incoming events and delivers all the events for a user to the same *Bolt*. For the events that need to keep consistency all the time for every reasoning engine, the states of the events are all disseminated to all *Bolts*.

Figure 23 illustrates a high level overview about the basic *Storm topology* in *Socialite*. Each event dispatches to a designated event channel, subscribed to by a listener *Spout*. The listener *Spout* gets a message, extracts the user name and creates a *tuple* that is sent to the respective processing *Bolt*. The groupings define to which *Bolt* the *tuple* is sent to. For *Device*, *User* and *Rule* messages, the *fields grouping* with *username* is used. *Relationship* and *Location* messages on the other hand are handled with *all grouping* because two users can be in the same location or can have a relationship together. Therefore, the relationship and location messages are immediately sent to all the processing *Bolts* to make the necessary

information available to all reasoning engine instances. Depending on the number of users, more instances of an implemented *Spout* or *Bolt* can be added dynamically during runtime by leveraging re-balancing of the topology mechanism supported by *Storm* [108]. The topology evaluation for the scalability with even distribution of users is discussed in Section 4.8.2.2

4.7 END USER EMPOWERED REASONING FRAMEWORK

Empowering users to program their smart spaces has been discussed in academic literature for decades. The authors in [32] report that autonomous technologies often leave users feeling out of control, without the possibility to adjust the level of autonomy of their home according to their needs. Newman [128] argues that end-user configurability is key in smart home applications and advocates sharing insights across a community of users.

With increased number of devices connected to the Internet and solutions from the industry such as IFTTT (“If This Then That”) [12], users create their own rules for the smart home devices as well as Web based services (e.g., weather service). However, the current solutions are limited to rules for a single device type rather than rules for each device capability. Furthermore, given the novelty of the Social Internet of Things (SIoT), social relationships are not considered in the end user programming and underlying reasoning engines.

We created *Socialite* to empower end users to create their own rules, which are used to reason about both devices and people in their social relationships in order to make automated decisions.

The features analyzed from the user survey in Section 4.2 are further classified into rule categories to develop the *Socialite* reasoning concept. The *Socialite* reasoning framework uses the *Socialite* semantic model to describe rules used in a production rule system namely Drools [18]. The semantic model provides a basic/low level knowledge representation while end-user created rules are high level knowledge representation for the reasoning in our system.

By synthesizing sets of guidelines for end user programming [65], we present a Web based trigger-action programming application that uses the APIs available in the *Socialite*

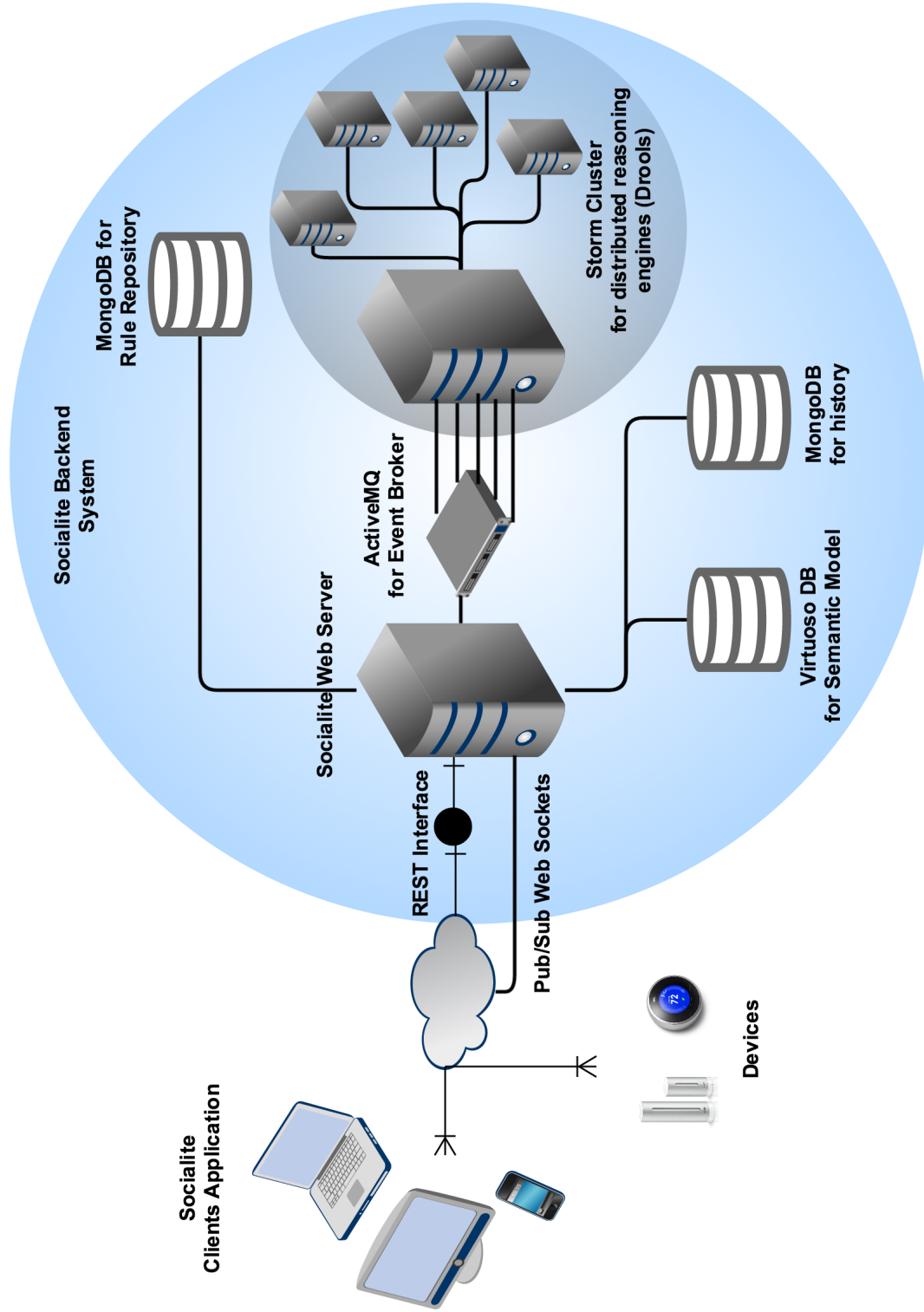


Figure 24: Event-driven distributed architecture

Server discussed in Section 4.6 for users to create their own rules running in the *Socialite Reasoning Framework*. The rule management REST APIs enables the client application developers to develop a light-weight end user programming tool without learning the domain specific languages used in Drools. This section first provide the *Socialite* reasoning framework concept, followed by the *Socialite* client application including the end-user programming application.

4.7.1 Socialite Reasoning Framework

The *Socialite* reasoning framework uses the semantic models for the basic/low-level knowledge representation (e.g., device and people), events/messages for asynchronous communication between the event sources and reasoning engines, and production rules for the high-level reasoning. Since our low-level knowledge is based on the ontology model with hierarchy, our reasoning supports both device-specific automation, and capability-based automation. Furthermore, the rules, leveraging social relationships and device capabilities will facilitate collaboration to efficiently share configuration and information even with devices from other users otherwise unknown to the user.

This section introduces the foundation of the production rule system, including knowledge representation and reasoning, then discusses the core reasoning concept with classification of rules and their attributes. The rule interfaces and translation mechanism to the domain specific rule language, and how rule management and sharing is done in the *Socialite* reasoning framework are also discussed.

4.7.1.1 Background of Reasoning Engines *Reasoning* is the formal manipulation of the symbols representing a collection of believed propositions to produce representations of new ones [49]. *Knowledge representation and reasoning* is concerned with how an agent uses what it knows in deciding what to do. The *reasoning engine* is the computer program that delivers *knowledge representation and reasoning* [49] functionality to the rule creator [18].

The *reasoning engine* has three components at high level, which are *ontology*, *rules* and *data*. The ontology is the representation model we use for our “things”. It could use records

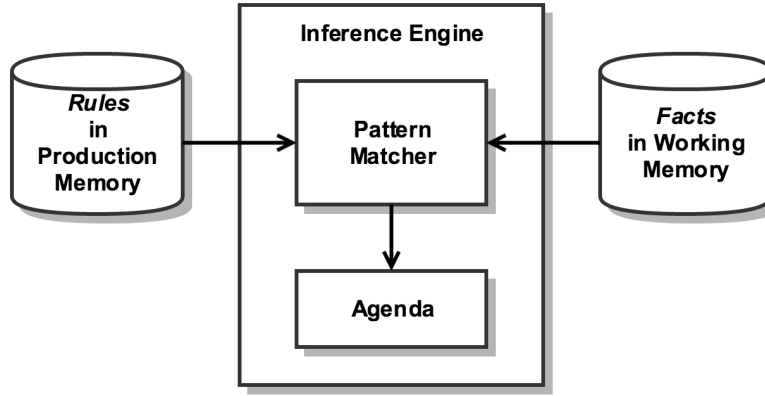


Figure 25: High level view of production rule system [18]

or Java classes or full-blown OWL based ontologies. In *Socialite* reasoning framework, Java classes with RDF/OWL annotations are used to formally represent objects that are needed for describing the domain of interests. The rules are used to perform the reasoning by an inference engine. Note that the *Socialite* reasoning engine does not include logical inference obtained from inference capabilities coming from the richness of the RDFS and OWL models. Rather, the *Socialite* reasoning framework uses rules of certain form called *production rules* as its representation of general knowledge [49]. Rules can be defined by system administrators during the development time (e.g., create *thriendships* between friends’ devices upon a new device registration or new friendship creation) and rules can be also created by end users to enable automation, complex event processing, context generation and service invocation in *Socialite*.

Drools is a *reasoning engine* that uses the rule based approach to implement an *expert system*. It is used for reasoning when processing the data with a set of rules to infer conclusion. *Drools* is categorized as a *production rule system* with a focus on knowledge representation to express propositional and first order logic in a declarative manner. Figure 25 illustrates the high level view of the *production rule system* with the core components.

A basic element of the rule model is “*when* condition is met, *then* actions will be exe-

cuted.” as illustrated in Code 4. Rules are stored in the *production memory* in the *production rule system*.

Code 4 Basic structure of the rule model in the production rule system

```

1      When
2          <condition>
3      Then
4          <action>

```

The *inference engine* matches *facts* that are information of objects in *working memory* against *rules* in the *production memory* in order to infer conclusions, which result in actions. The process of matching the new or existing *facts* against *production rules* is called *pattern matching*. *Drools*’ *pattern matching* implements the *rete* algorithm [77] and is handled by the *inference engine*. The *inference engine* is able to scale to a large number of *rules* and *facts*.

A system with a large number of *rules* and *facts* may result in many of the *rules* being true for the same *fact* assertion. These *rules* are said to be in *conflict*. The *agenda* is a collection of activated *rules* and it is responsible of managing the execution order of the *conflicting rules*. When the rules are being fired after *pattern matching* process, *Drools* selects one rule from the *agenda* and executes its *action*. If a *fact* in the *working memory* is changed as a result of the current *rule* execution, other *rules* can be activated or deactivated. This continues until the *Drools agenda* is empty, that is there are no more *rules* to be fired. The order in which *Drools* fires the *rules* can be configured with *conflict resolution strategy*. *Drools* provides *salient* (or priority) and *LIFO* (Last In First Out) as *conflict resolution strategies*.

There are two methods of execution for a rule system: *forward chaining* and *backward chaining*; systems that implement both are called *hybrid chaining systems*. *Drools* provides *hybrid chaining*, both forward and backwards. *Forward chaining* is “data-driven” and thus reactionary. *Backward chaining* is “goal-driven”, meaning that it starts with a conclusion that the engine tries to satisfy. The *Socialite*’s reasoning is done with *forward chaining* because most of *rules* are reactionary upon the evaluation results from the inserted *facts* used in the rule evaluation *condition*.

A *reasoning engine* has two different modes of operation: a *stateless* or a *stateful knowledge session*. A *stateless session* has its own isolated data/state that is disposed at the end of the invocation, whereas *stateful session* are longer lived and allow iterative changes over time [18]. In *stateless knowledge sessions* data is not saved; it can be processed once and after that the data cannot be retrieved again. On the other hand, utilizing a *stateful knowledge session* on the other hand allows the system to process the data, detect changes over time, calculate average, minimum and maximum, and so forth. *Temporal reasoning* is possible using the *stateful knowledge session*. The *Socialite* system configures *Drools* with *stateful knowledge sessions* because the rule category includes *temporal reasoning*.

4.7.1.2 Socialite Reasoning Concept This section addresses the core concept of the reasoning framework in the *Socialite* system, including the classification of rules and attributes in each rule. The rule categories are driven by the analyzed results from the user survey in Section 4.2.3 and our system design decision.

Mapping Feature Categories from User Survey to Rule Categories

After the feature categorization of all answers from the user survey discussed in Section 4.2, nine feature categories were mapped into rule categories and other functions (see Figure 26). Two types of user roles are considered in rule creation: end users and system developers. The rule categories for end users include automation, context generation and service invocation. The rule category for explicit relationship management is assigned for system developers, because the *Socialite* system aims to provide default rules for explicit relationship management based on our concept of social relationships. In addition, remote access and control feature is assigned as a basic client application feature since it does not require any rules to enable this feature. The rest of two features (goal and others) are mapped to other technologies with new event types that requires different technologies (e.g., security and privacy, data mining) .

The automation rule category combines device/capability, preference and context based automation where the action in the rule description can be expressed based on the semantic

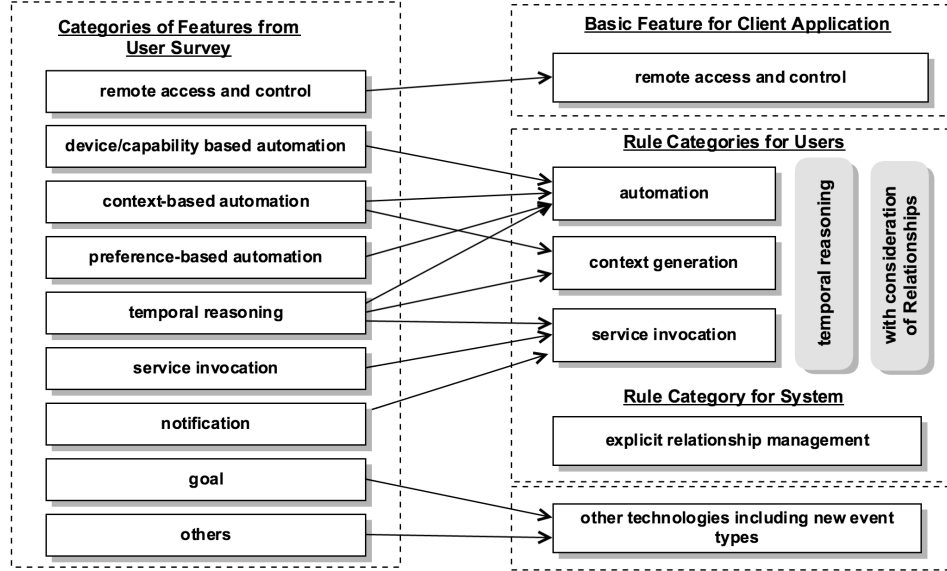


Figure 26: Mapping categorized features from the user survey to rule categories and other functions

model. On the other hand, in order to enable context based automation, we introduce the context generation rule category where the user can specify what leads to a user's context. Obviously there exist other technologies to determine a user's context (e.g., applying the Bayesian approach [84]), but this dissertation addresses user-defined context and its reasoning based on the knowledge represented as a rule description.

The notification and service invocation features are categorized into the same rule category, because the examples in the notification feature are mostly related to the display type of devices, which are not part of our semantic model and require a service call to follow the manufacturer's notification APIs.

Temporal reasoning and relationships can be used for all the rule categories to express the condition and/or the action in a rule description.

More detail about each rule category are explained with examples as below.

Classification of Rules

- **Automation based on device capabilities, context, preference and temporal reasoning**

The *Socialite* reasoning engine supports creating rules to specify automation, that is, the change of a set of device capabilities is triggered based on a specific condition that user defines. Unlike conventional rules used in other smart home systems, the automation rules in the *Socialite* leverage not only a specific device type but also the *device capabilities*.

Furthermore, the devices in the new social relationships, such as *thriendships* are used in order to express the *condition* and/or *action* in the rule. As for *action* expressions we limit the devices to the ones owned by the user as default. Depending on the access control policies, the devices used in the rules can be selected differently in the future. Although the current *Socialite* system does not support an access control mechanism yet, the access control mechanism developed for the smart home gateway (see Section 3.3) can be extended for the *Socialite* system with consideration of new social roles.

In addition, the *context* defined by the user can be used in expressing *conditions* in the rules. An example of an automation rule, both with device capability and context shown in Code 6 is “*When Jenny’s living room’s temperature is greater than or equal to 80 °F and Jenny is at home, then set Jenny’s air-conditioner’s target temperature to 75 °F*”. The preference that is accessible via the device semantic model can be used both in the *condition* and *action* expressions to get/set the target value/state in a rule.

- **Context Generation**

As shown in the analyzed results from the user study (see Section 4.2), users expect to have automated decisions, not only based on the device status change itself, but also with a higher level situation that can be expressed with the *Socialite* semantic model, information obtained from external services (e.g., weather) and temporal reasoning. The definition of contexts is varied amongst researchers. One of the highly received context definitions is that “Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant

Code 5 Context generation rule example

```
1   declare Context
2   contextId : String
3   end
4
5   rule "context_generation_example"
6   when
7   (
8       $r1 : Device ( deviceId == "jenny_TV_LivingRoom" ) and
9       DeviceProperty ( property == "OnOffState", propertyValue == "On" ) from
10      $r1.deviceProperties
11  )
12  (
13      $r2 : Device ( deviceId == "jenny_Light_LivingRoom" ) and
14      DeviceProperty ( property == "Brightness", propertyValue < "70" ) from
15      $r2.deviceProperties
16  )
17  then
18      insertLogical ( Context ( contextId = "jenny_watchingTV" ) );
19  end
```

to the interaction between a user and an application, including the user and applications themselves [67].”

Instead of extracting the context automatically by the system, our solution allows end users to define their own context by using end-user programming. Unlike other smart home systems where the context is a part of the ontology model within the system [166], our system proposes context as a rule generated by a user and sharable with others depending on their privacy settings on the rules. The user-defined context that is inferred by the reasoning engine can be further used by automation rules discussed in the first rule category. The following rules (Code 5 and Code 6) show how a context is defined and how the inferred context is used for another rule in the automation category.

- **Service Invocation** *Socialite* allows *actions* in the rule to be expressed with a service invocation. Services can be provided internally by the *Socialite* system or externally from third party service providers. The services can further use the relationships of the devices and users, a feature that is not yet possible for current practice of the Internet of Things. For example, a service that leverages devices in *kinship* can be used to provide

Code 6 Automation example with user-defined context

```
1 rule "when Jenny is watching then turn off Jenny's lamp"
2 when
3     Context (contextId == "jenny_watchingTV")
4 then
5     setDeviceAction ("jenny_lamp", "deviceCapability="OnOffState", "
deviceAction="Off");
6 end
```

repair solutions by analyzing the repair history of the devices in *kinship*, which had the same error before. Since the *Socialite* semantic model employs the hREST ontology [105] from W3C to represent the REST service interface, any services registered to the *Socialite* system can be expressed in the *action* in the rule. When a rule is fired, a service invocation is called by executing an internal function implemented in the *Socialite* reasoning framework.

- **Explicit Relationship Management** *Socialite* supports specifying a rule to infer explicit relationships. As an illustrated example, let us assume that User 1 owns Device A and User 2 owns Device B. If User 1 and User 2 establish a friendship, or if another device is added to a user's roster of devices, the rule engine may trigger a rule to explicitly create and store new *thriendships*. In addition to the rule for these static relationships, the *Socialite* supports the dynamic aspect of the relationship such as *co-location*. The location of mobile devices (e.g., smart phones) is dynamic in nature; devices may disappear from an area and re-appear in a different environment. The *co-location* relationship rule is triggered based on a device's location. The derived relationships, such as *thriendship* and *kinship* as well as dynamic changes of the *co-locationship* are created by default in the *Socialite* system while *ownership* and *friendships* are controlled by the user. The explicit relationship management rule is configurable to be active or inactive for users who have privacy concerns and do not want the system to create relationships automatically (e.g., automatic creation of co-location relationships with people or devices that a user does not know).

Temporal reasoning in the rule description

The *condition* and/or *action* of all of the rule categories in the *Socialite* system can express temporal aspects of the object with Complex Event Processing (CEP). CEP tracks streams of events (facts in a row) that are inserted into the reasoning engine and detects a specified temporal event in real time. In detail, CEP enables evaluation of whether *event1* happened before or after *event2* or if events happened in a specific time frame. In order to support CEP within a reasoning engine, the Drools engine is configured in a stateful session.

The use of CEP in the *Socialite* aims to identify events in real time over relatively short time periods (less than 30 minutes) either explicitly by specifying the event expiration offset in the rule or implicitly by analyzing the temporal constraints in the rule. Operation related to event detection over long periods of is not addressed in CEP based rule in our system. An example of rules from the user survey is “if my heater’s performance is lower than the heater in my friend’s home, then notify me with possible solutions”. This would be done with other technologies leveraging the harvest data in the device history repository, for example applying abnormal detection algorithms [54]. The rule relevant for this example is rather categorized as a *service invocation* in our rule categories, because it requires to access the device history for long periods which is not stored in the production memory in Drools, but stored in the device history repository.

Relationship used in the rule description

One novelty of the *Socialite* reasoning framework is that the relationships can be used in all categories of the rules. A rule can specify a specific device by selecting one from the filtered devices and relationship type, or specify a set of devices with relationship type and capabilities.

Attributes in Rules

The following four attributes are considered in the rule description of the *Socialite* system. These attributes are used for the rule operation and management in the system as well as sharing with others.

- **Sharing attribute:** We expect that the users of the *Socialite* system share their rules to address common goals collaboratively. Towards that end, the *Socialite* reasoning framework supports the sharing attribute. The possible values for the sharing attributes are 1) private, 2) public or 3) constrained to certain relationship types such as friendship and thriendship.
- **Event attribute:** A rule is triggered by an external event that is notified to the *Socialite* reasoning engine in the form of a message from the connected devices, users or relationships as discussed in Section 4.6.3. A time event is modeled as a possible value for the event attribute. The state-of-the-art reasoning engine such as Drools provides the time event within the reasoning engine itself, the *Socialite* distinguishes a time event from other events outside the reasoning engine. The time events are further modeled into two perspectives: 1) periodic time event and 2) a time event at a certain point. When a rule from the end user programming is translated into the syntax of the Drools domain specific language, these time perspectives are embedded in a rule description and time events are managed by Drools.
- **Activeness attribute:** A rule is modeled to support activeness attribute so that users can manage their rules to be active or inactive depending on their usage patterns. A rule can be active from now, inactive from now, or active only a certain time period. Operational status based on the activeness can be specified in a higher level rule. For example, a rule can specify that one rule is only active after it meets a certain condition (e.g., the number of execution is greater than a threshold).
- **Goal attribute:** The *Socialite* introduces the goal attribute for further collaboration and information sharing. A set of rules can have a common goal attribute such as energy saving, or securing homes.

Rule Triggers

Depending on the event attribute values in *Socialite*, external events or schedulers are currently implemented as rule triggers in the *Socialite* framework. An external event is initiated when an object is inserted into the reasoning engine.

For example, if a device object is changed, the *Spout* listener, which subscribes to that change, inserts the updated device object into the reasoning engine. As it enters the engine,

a fact is recorded and all the rules that have this device model in the rule condition are evaluated. If the condition of a rule matches the fact in the working memory, the rule is ready for execution.

In case of the timing event, rules can be scheduled at certain times or periodically (e.g., with a *cron* expression). The *cron* syntax is expressive and enables precise scheduling from one second granularity to any time period (e.g., a rule can be fired only once per year).

To give the user more possibilities to specify more advanced rules, time restrictions on the rules are considered. This includes allowing the user to specify a time frame in which the rule is active; the rule would be inactive outside that time frame.

Drools already supports two specific keywords related to this issue, namely *date-expires* to let a rule expire at a specific date and *date-effective* to activate a rule when the specified time comes. These two keywords are in the date level granularity, not in the hour level granularity.

In order to overcome the above limitation of the Drools reasoning engine, the *Socialite* framework provides its own *TimeFrame* class to allow for the expression of the current time including hours, minutes, months, even the weekday. By doing so, our system provides fine-grained scheduling of the rules and also supports with the activeness/de-activeness of the rule attribute.

Conflict Resolution Management

Rules in the system can conflict with each other since rule creation time, but actual detected conflicts of rules can happen when a fact is inserted to the reasoning engine. The reasoning engine detects possible conflicts and decides the conflict resolution strategy. For example, consider a thermostat that accepts two rules, which are 1) “if a user is away, then set the target temperature value to 60 °F” and 2) “if a user is away, then turn off all the devices.”

Conflict resolution is required when there are multiple rules on the agenda, which is responsible of managing the execution order of the conflicting rules in Drools. The reasoning engine needs to know in what order the rules should fire (for instance, firing *rule A* may cause *rule B* to be removed from the agenda). The default conflict resolution strategies employed by Drools are: *Salience* (or priority) and *LIFO* (last in, first out).

The most visible one is *salience*, in which case a user can specify that a certain rule has a higher priority than other rules. In that case, the rule with higher salience will be preferred. LIFO priorities are based on the assigned *working memory* action counter value, with all rules created during the same action receiving the same value. The execution order of a set of firings with the same priority value is arbitrary.

Rule Execution in Emergency Situations

Using the *Socialite*'s rule concept, an emergency situation, such as fire, can be handled efficiently. An emergency context would be triggered after two or more events from either devices (e.g., smoke detector) or people (e.g., alert an event to neighbors). Once the emergency context is confirmed, prioritized emergency rules can be activated. Examples include “If a fire occurs, unlock all doors and windows” or “If fire occurs, notify police and neighbors with the user’s location.” Once the emergency is cleared or the activation time period of the rules has passed, emergency rules can be deactivated.

4.7.1.3 Rule Transformation to Domain Specific Language A user-specified rule is created via the end user programming tool. We aim to enable the end-user programming tool developers as well as end users to develop the tool or create rules without detailed knowledge of the domain specific language used in *Drools*. The Drools’ rule language is based on declarative programming, and therefore shifting paradigm to a declarative rules style and learning how to write the rules properly and effectively can be time consuming for the client application developers and the rule creators.

In order to address this concern, the *Socialite* server provides rule management interfaces to the end user programming tool so that the client application (which includes end-user programming) can be developed by using REST APIs. The *Socialite* server includes a rule translation mechanism that translates the rule data payload in JSON format to the domain specific language used in the Drools reasoning engine.

The JSON payload in the rule creation REST interface (an example is in Code 7) is transformed into Java objects using an existing library such as Google gson. The rule translator uses the Drools APIs to generate the Drools rule objects, which can be put into

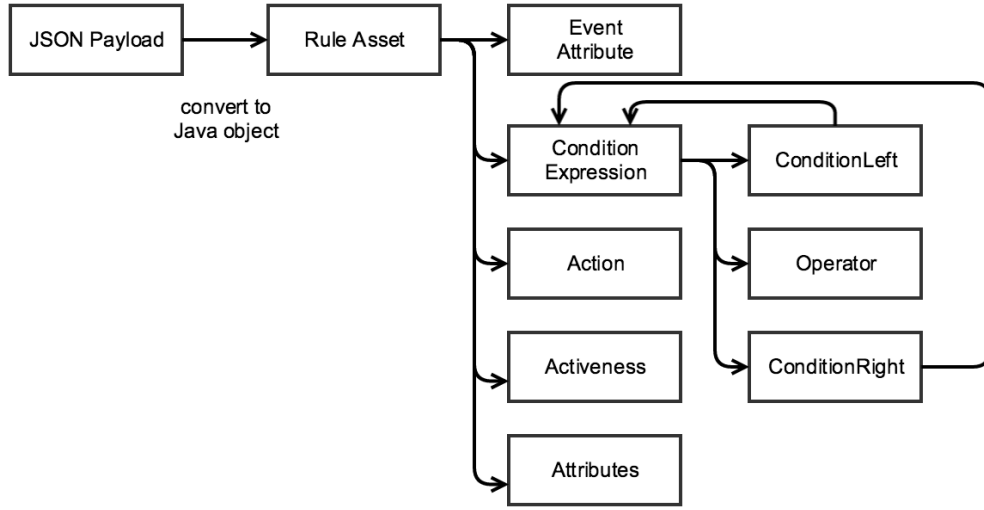


Figure 27: Intermediate common data models between the data payload in REST APIs and Drools domain specific language

the reasoning engine. Figure 27 represents the objects in the rule model, which are used as an intermediate common data model between the data payload in JSON format and the Drools domain specific language.

4.7.1.4 Rule Management and Sharing If people and devices are interacting with each other through explicit relationships as proposed in the *Socialite* system, we expect that people are more socially interactive when they use their connected devices. The user survey discussed in Section 4.2 indicates that one of the main reasons why people share their information from devices is to help others. Therefore, sharing the rules created by an individual user can promote their motivation to participate in the *Social Internet of Things* and ultimately achieve a common goal with sets of rules classified into a same goal (e.g., energy saving in a city).

This section discusses how we enable users to share their rules and how these rules are instantiated by other users' environment. Figure 28 illustrates the rule sharing concept.

Code 7 Example of JSON payload

```
1 {
2   "Rule":{
3     "ruleType":" Automation",
4     "name":" rule_example1",
5     "username":" jenny",
6     "ruleId":" jenny_rule_example1",
7     "summary":" This is an example rule for the description of the JSON rule
format.",
8     "privacyAttribute":" private",
9     "goals":[
10      "EnergySaving"
11    ],
12     "eventAttribute":{
13       "type":" event",
14       "eventType":" UpdateDevice"
15     },
16     "activenessAttribute":{
17       "activeness":" active"
18     },
19     "conditionExpression":{
20       "condition1":{
21         "model":" Device",
22         "modelId":" jenny_Thermostat_LivingRoom",
23         "modelType":" Thermostat",
24         "property":" TemperatureF",
25         "operator":" >=",
26         "value":" 80"
27       },
28       "condition2":{
29         "model":" Context",
30         "modelId":" jenny_atHome"
31       },
32       "operator":" &&"
33     },
34     "actions":[
35       {
36         "model":" Device",
37         "modelType":" AirConditioner",
38         "modelId":" jenny_AirConditioner",
39         "action":" TargetTemperatureF",
40         "operator":" =",
41         "value":" 75"
42       }
43     ]
44   }
45 }
```

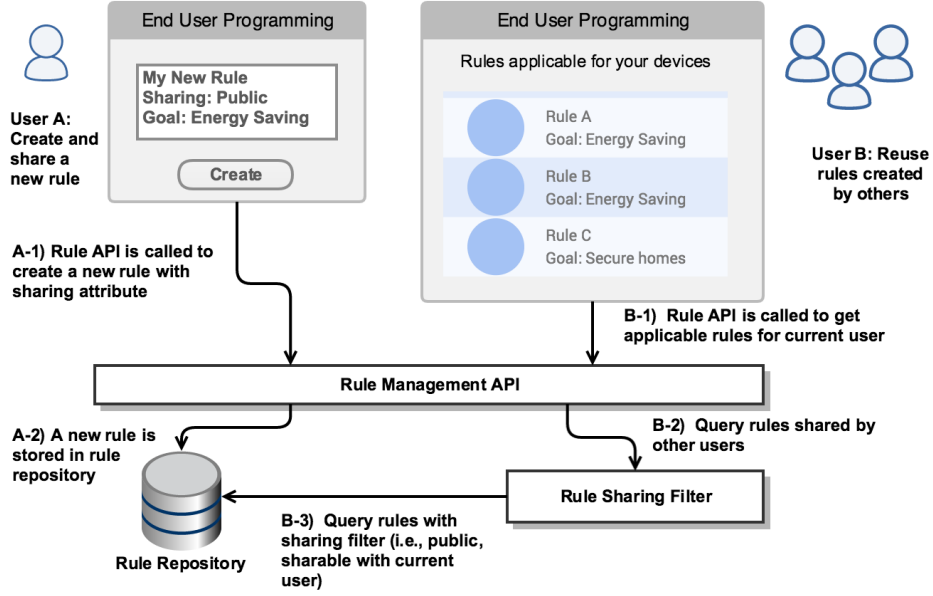


Figure 28: Rule sharing concept

When a rule is created, sharing attributes should be specified with one of these values: private, public, sharing with relationship types (i.e., union or conjunction). When a user queries shared rules created by other users, the *rule management* component takes into consideration sharing attributes, device capabilities and goal attribute used in rules and returns only possible rules the user can apply to the user's current system. The device capabilities and the goal attributes are optional to use as filters for querying the sharable rules because users could be interested in other rules that require new devices currently not owned by users.

To measure the goal achievement, a new method to calculate each rule's contribution to a goal. The *Socialite* system has not realized any solutions in this dissertation, however, awareness, visualization, sharing of their status of achievement and gamification can motivate users participate in a common goal such as energy saving [115, 56]. For example, if energy saving is a goal, then we can consider different ways to visualize their achievement: ranking of participants in the same city (co-locationship) or in friendships with their total energy

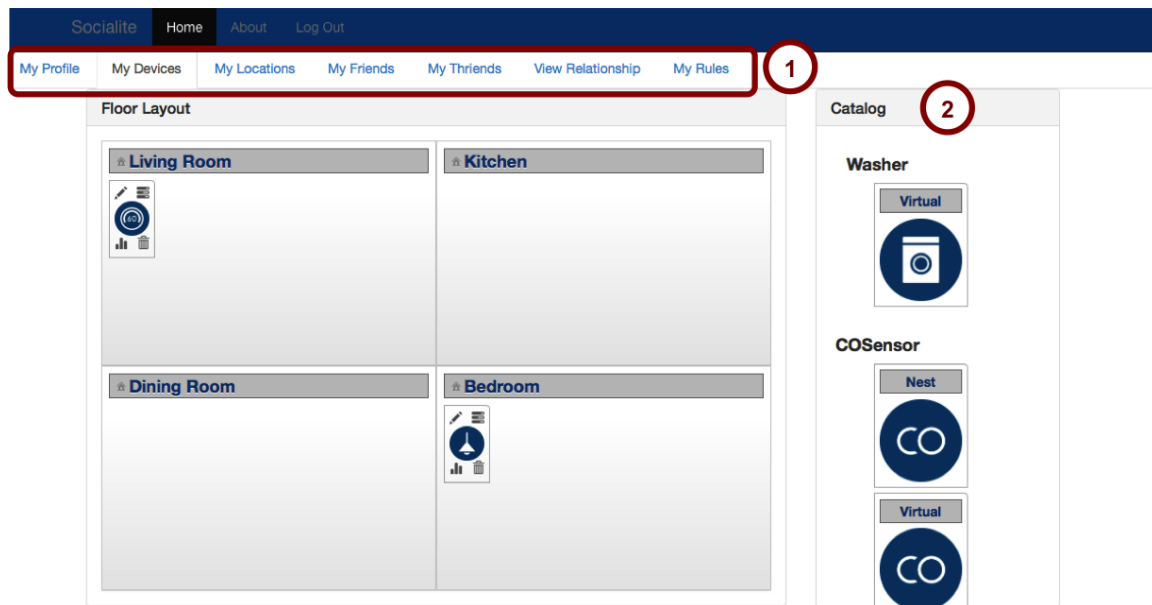
consumption from the smart meters and/or simulated energy consumption based on the device usages. If a goal is securing home, the goal achievement can be done by activating relevant rules to help secure neighbors' homes, or actions taken by the participants when an alarm goes off.

4.7.2 Socialite Client Application

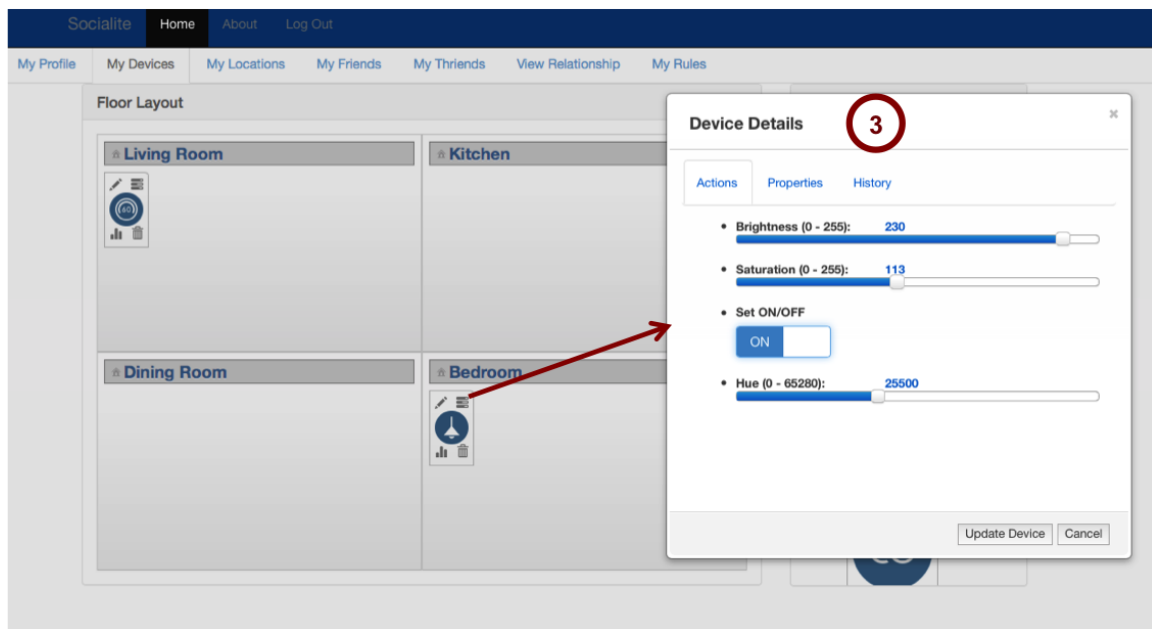
The *Socialite* client application is a Web based application using Backbone.js, which is a light-weight JavaScript library with a REST Web Service and is based on the *model view presenter* design paradigm [134]. We selected a Web based application because potential users would use different mobile devices and computers when they access the existing social networks site, as we found in the user survey. The Web based application can be accessible from any type of devices. Obviously the client application can be extended with mobile applications in the future.

4.7.2.1 Features Figure 29a shows all menus available in the *Socialite* Web based application. As shown in ① in Figure 29a, user profile management, user's owned device management, user's friends management, user's owned devices' thriends management, and rule management are realized in the *Socialite* client application. The Figure 29a is a screenshot of when a current user clicks "My Devices" menu from ①. It shows that two devices are registered in Living room and Bedroom. If the current user wants to register a new device, the user clicks one of devices from the device catalog in ②, and drag and drop to one of the rooms in the left pane. Clearly, the room layout and other user interface can be further enhanced with other available libraries, but it is beyond of the scope of this dissertation.

When a user clicks right top corner of icon representing one of the registered devices, the user can monitor and control the selected device by interacting with the pop-up window as shown in ③ in Figure 29b. The history of the selected device is visualized as a graph when the user clicks the left bottom corner of the device icon on the room layout. A device can be removed by also clicking the trash can on the right bottom of the device icon. The left top corner of the device icon is used to edit the device information.



(a) Basic features



(b) Remote control of a lamp

Figure 29: *Socialite* Web based user interfaces

Push notifications originated by the server are sent to the client application through web sockets and managed by the publish and subscribe service in the server side.

All operations originated by the client use the *Socialite* REST APIs provided by the *Socialite* server.

4.7.2.2 End User Programming for Rules The *Socialite* end-user programming tool is represented to the user as a trigger-action programming, which is one of the most common formats in the academic literature that matches with users’ mental model [65, 68, 163].

Figure 30 represents the main UI view of a rule creation. The devices listed in ① are devices that satisfy the filtering options from ① (*relationship types*) and ② (*device properties* or *contexts*). If the user drags and drops one of the devices or contexts listed in ③ (*Trigger* panel) to ⑦ (*IF* panel), then a pop-up window is shown to allow the user to add a condition (e.g., temperature >80 °F). The user can add as many devices and contexts from ③ as s/he wants to express more conditions. The default operator among condition is conjunction.

In the panel ⑥ (*Actuator* panel), all user’s devices with action capabilities and services are shown as a default. The user can select a different relationship type in ④ or filter devices with a certain action capability or services in ⑤. The user can select one of devices or services and drag it to ⑧ (*THEN* panel). The user can set the action value by using the pop-up window (⑨) with a default action pre-selected from ⑤. For the service actuator, the user can select relationship types provided by the service. For example, a “repair solution query service” may ask a user to select kinship or/and thriendship as parameter values used for the implemented service. The user can also create a context instead of device action or service invocation, by clicking the *save context menu* in ⑤.

Once the user finishes the condition and action expression of the rule, then the user clicks *save button* in ⑩, which will call a *Socialite* server API to add the new rule for the current user after providing the rule meta information, such as the rule name and description.

The current implementation of the end user programming application enables the device capability as well as device type based automation, context generation, context based automation, and service invocation.

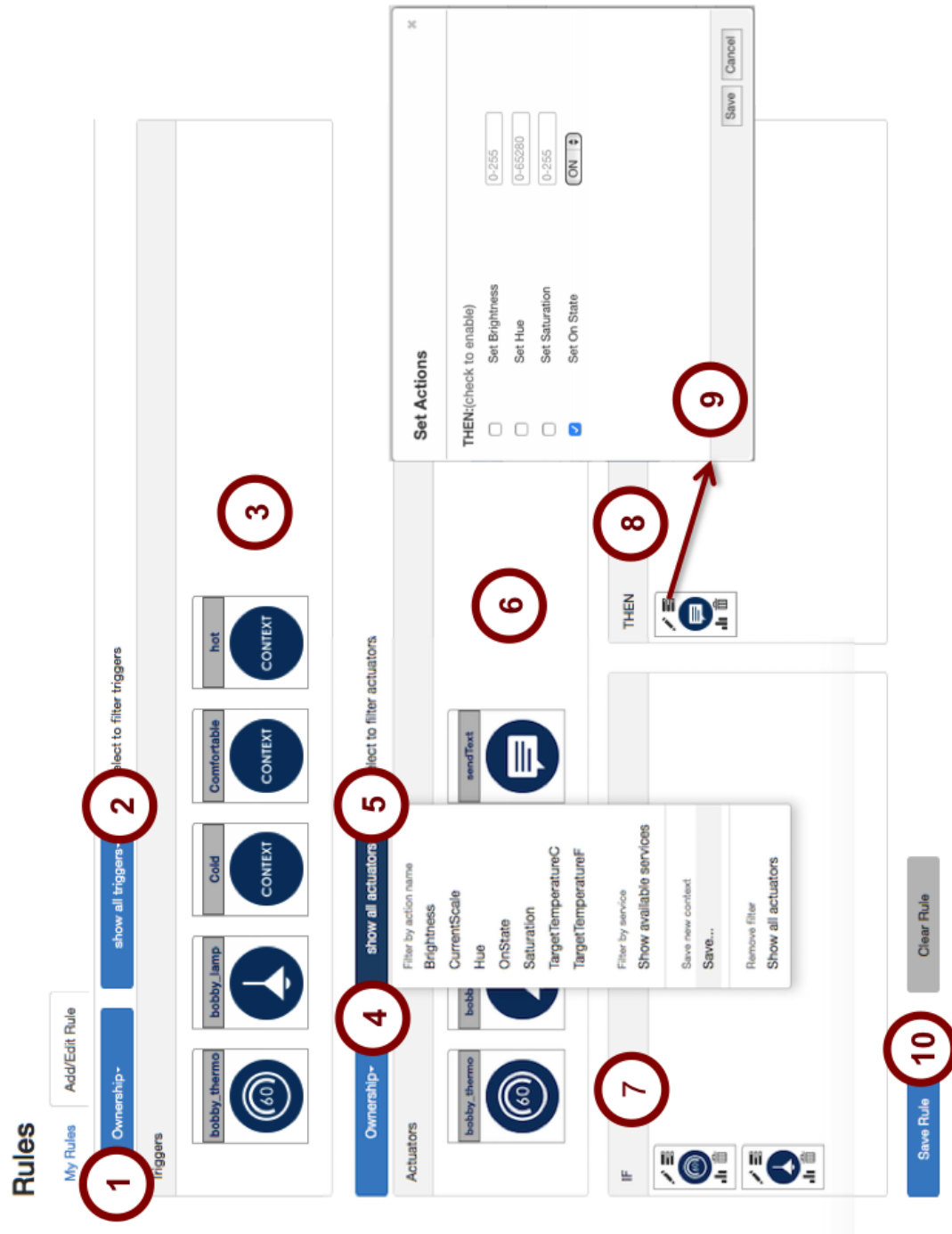


Figure 30: End-user programming user interface

4.7.2.3 Architecture Overview The *Socialite* client application is a light-weight Web-based application that communicates with the *Socialite* server via REST APIs. As explained earlier in this section, we employed Backbone.js as a basic framework. Therefore, the architecture design is based on the *model view presenter* architectural pattern. Figure 31 illustrates the *Socialite* client architecture overview with a rule creation user interface.

The client UI *view* represents HTML documents together with templates, which are markup that can be used to create different reusable copies of that markup but populating each component with different data. The *model* represents the data model such as rule, device, relationship, user, etc. The *presenter* (e.g., rule presenter, device presenter) uses the observer design pattern [78] to subscribe to changes from the model.

The *Socialite* end user programming support device/capability based automation, context based automation, context generation and service invocation with consideration of relationships. Although the server APIs are available, the existing end user programming has not yet implemented preference based automation and temporal reasoning in the user interface.

4.8 SOCIALITE IMPLEMENTATION: PROOF OF CONCEPT

This section provides how the proposed *Socialite* architecture and core concepts are realized in a real system, which integrated real devices and is currently deployed in the Amazon Web Service EC2 instances [8]. It also discusses an evaluation of the propose architecture with respect to interoperability, scalability and extensibility.

4.8.1 Implementation

The *Socialite* framework implements the concept discussed in the previous sections, resulting in a real-world test-bed that integrates devices from different manufacturers deployed in multiple homes. Note that *Integrity* (see Chapter 3) targets for the integration of devices using different communication protocols and serves as a unified *home gateway*, while

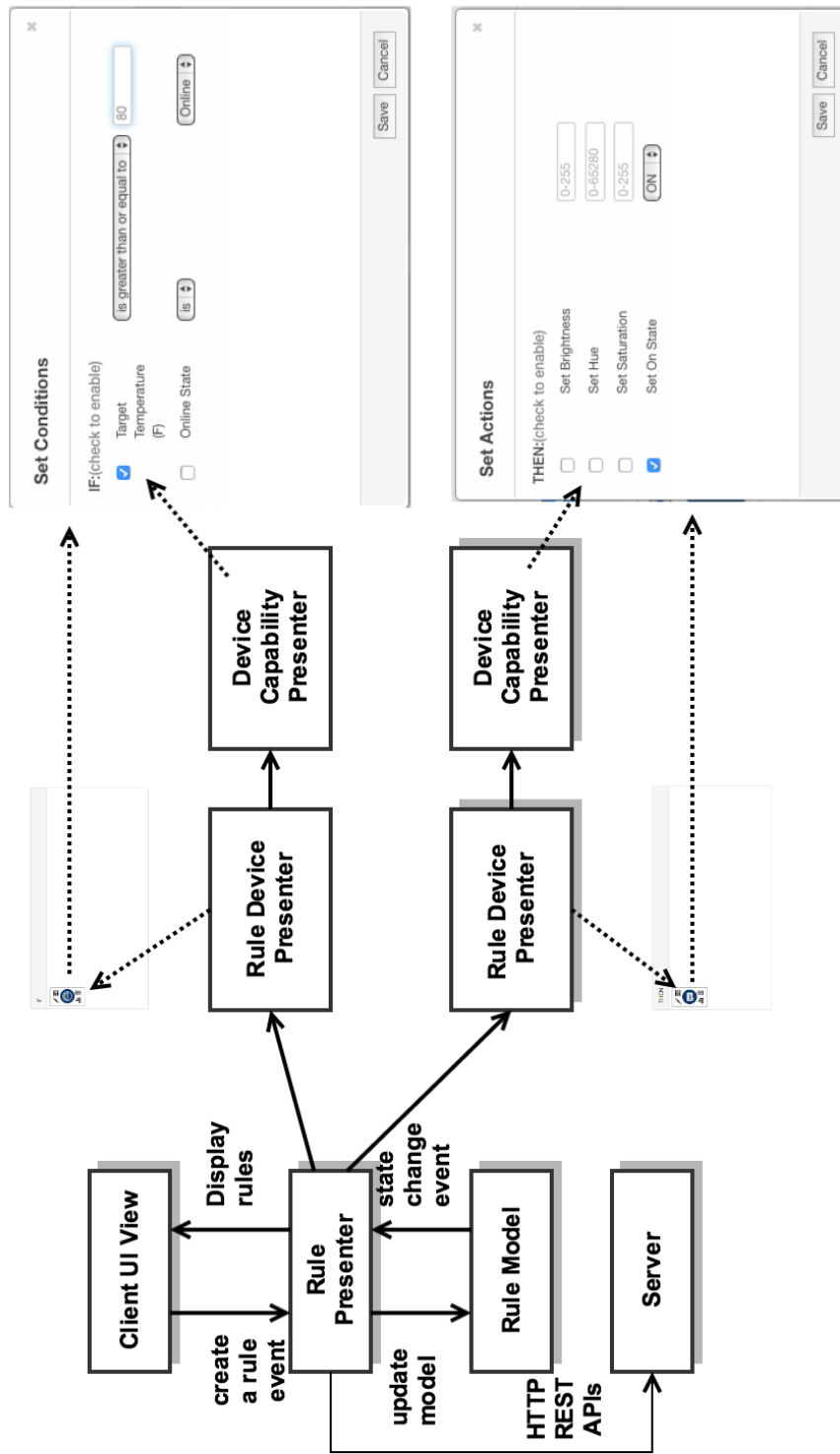


Figure 31: *Socialite* client architecture overview with a rule creation example

Socialite addresses the “social” and “physical” integration of different *users*, *devices* and *home gateways* (with many devices attached) in the cloud. Currently, *Socialite* provides support for devices with public APIs, such as Nest Thermostat and Nest Smoke Detector [14], NetAtMo Weather Station [127], Philips Hue [15], and Jaw Bone [13]. These devices have been deployed in a real home and a lab (located in Pittsburgh, PA, USA) and can be accessed through the REST Web service API provided by *Socialite*.

Besides the support for real devices, the *Socialite* framework also provides the implementation of virtual devices for existing devices that lack API support, for future devices currently under development at a start-up company and for simulation of devices. Such devices are useful when real devices are not available, and one would like to test some specific aspect of the framework such as extensibility.

The device types implemented in *Socialite* system include CO2 sensor, humidity sensor, smoke detector, motion sensor, noise sensor, temperature sensor, anemometer, blind controller, contact sensor, cooker, coffee maker, damper controller, dish washer, door lock, door controller, fridge, IP camera, oven, power meter, pressure sensor, siren, television and thermostat.



All software is deployed to multiple instances on the Amazon Web Service for the evaluation. Figure 33 illustrates the deployment view of the *Socialite* instances for the proof of concept.

As discussed in Section 4.7.2, management of users, devices, relationships and rules are possible through the *Socialite* client application. The client application also includes remote access and control and end-user programming for the various rule creation.

The *Socialite* demo video is available in [97] as of the publish date of this dissertation. The video demonstrates important features by using two devices: a Philips Hue lamp and a Nest thermostat.

First shows a remote access and control of a Philips Hue lamp by turning on and off and changing the color of the lamp through the client application. Then it shows a rule creation for an automation rule, namely “If the lamp is on, then set the thermostat’s target temperature value to 70 °F.”

The next scene is to create three different contexts: 1) “If the temperature is ≤ 60 ,


swagger


Explore

devices : Rest interface to access devices
Show/Hide List Operations Expand Operations Raw

relationships : Various relationships for Socialite
Show/Hide List Operations Expand Operations Raw

POST
/relationships
Registers a new relationship

GET
/relationships
Returns all relationships

Implementation Notes

list of all relationships

Response Class

Model Model Schema

```

Relationship {
  relationshipId (string, optional),
  objectId (string, optional),
  subjectId (string, optional),
  subjectThing (SioThing, optional),
  subjectType (string, optional),
  objectType (string, optional),
  relationshipType (string, optional),
  objectThing (SioThing, optional)
}

```

Response Content Type application/json

[Try it out!](#)

GET
/relationships/{subjectId}
Returns all relationships for the specified subject

DELETE
/relationships/{relationship_id}
Delete a relationship permanently

GET
/relationships/{subjectId}/{relationship_type}
Returns relationships for the specified subject of the specified type

users : human user information
Show/Hide List Operations Expand Operations Raw

rules : rules specified
Show/Hide List Operations Expand Operations Raw

locations : Rest interface to get all locations registered
Show/Hide List Operations Expand Operations Raw

services : Rest interface to access services
Show/Hide List Operations Expand Operations Raw

Figure 32: Dynamically accessible *Socialite* APIs

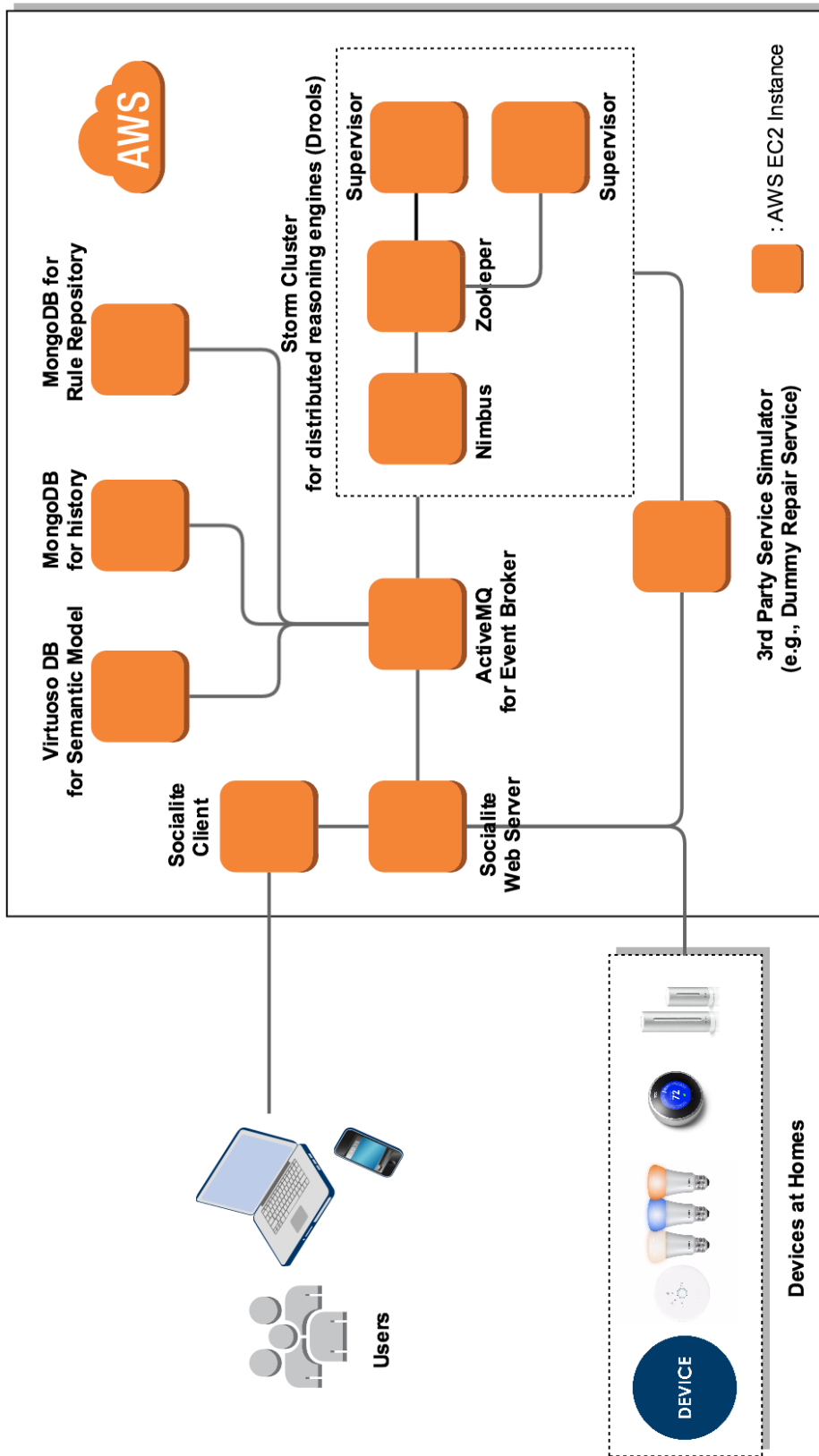


Figure 33: *Socialite* deployment view on Amazon Web Services (AWS)

then context is *cold*.” 2) “If the temperature is between 60 °F and 75 °F, then context is *comfortable*.” and 3) “If the temperature is ≥ 75 °F, then context is *hot*”. By using these three contexts, three context-based automation rules are created: 1) “If the context is cold, then set the lamp color to blue.”, 2) “If the context is comfortable, then set the lamp color to green.”, and 3) “If the context is hot, then set the lamp color to red.”

Then the video shows the context-based automation in a real environment by changing the temperature value in the Nest thermostat. The lamp color is changed depending on the context, which is determined by the temperature.

In the next scene, the user adds a new friend (Jenny), and shows that Jenny’s devices are added as *thriends* and displayed in *My thriends* menu.

In the next scene, the video shows a rule creation with *thriendship* relationship, which is “If Jenny’s lamp is on, then turn on my lamp.”

In the last scene, the video shows another rule of “If an error occurs in my device, then call the service that sends a text message to my friends in *co-location* relationship.”

4.8.2 Evaluation of Architecture

Our architecture supports *interoperability*, *scalability* and *extensibility*. This section includes evaluation of our architecture with respect to these three non-functional requirements.

4.8.2.1 Interoperability Evaluation Interoperability is empirically evaluated by integrating the real devices to the *Socialite* system.

Interoperability is supported in the *Socialite* system by providing a uniform access through common semantic models to different implementation of functions to interact with various APIs from different manufacturers. The device authentication model has a list of authentication parameters with two data properties of key and value that supports various authentication mechanisms required by the manufacturers.

An example of interoperability from the devices used in the proof of concept is humidity sensor from NetAtMo and Nest Thermostat. Both of them represents a same device model for humidity sensor, but APIs from NetAtMo is completely different from the ones from Nest.

Obviously they provide different URLs and payload to get the value of the humidity sensor. Furthermore, the communication styles are also different for these two manufacturers. In detail, NetAtMo does not send change in the sensing value to the *Socialite* server, rather the *Socialite* server needs to call NetAtMo API to get the current value periodically. On the other hand, Nest allows the *Socialite* server to subscribe to the change in value. Regardless of different implementation on how to get the humidity value from these two different devices, the common semantic model in the *Socialite* server always presents the same data model and provide a uniform access to different devices.

NetAtMo and Nest use OAuth2 to authenticate the device. Therefore we reuse the same software module for authentication but with different parameter values. However, Philips Hue APIs does not support OAuth2 authentication mechanism. Therefore, to access Philips Hue APIs we use different authentication parameters which are the username for the bridge (gateway) that locally connects to the lamps.

4.8.2.2 Scalability Evaluation The *Socialite* aims to achieve scalability by distributing a number of reasoning engines over multiple *Bolts* (see Section 4.6.4.2) installed on different nodes to process reasoning in parallel.

The *Socialite*'s event-driven architecture that uses an implementation (ActiveMQ [155]) of Java Message Services (JMS) and a real-time distributed computing engine (Apache Storm [159]) enable the *Socialite* system to be scalable; these two components make it scalable because of asynchronous communications and parallel processing of messages from devices, users and services.

Apache storm is designed to scale out by adding more nodes to a cluster. Therefore, we evaluate scalability for even distribution of tasks with the *Socialite* Storm topology discussed in Section 4.6.4.2.

Topology Performance Evaluation

In order to evaluate the scalability with even distribution of users, the distribution testing is conducted by varying the number of *User objects* while keeping the number of *Bolts* fixed at 10. A *Spout* produces a specific number of different user names for each test case and sends them one by one via *fields grouping* to 10 different *Bolts* that count the number of

test case	1	2	3	4
number of users	100	200	1000	10000
expected number of users per bolt	10	20	100	1000
maximal difference	4	6	9	21
maximal difference in percent	40%	30%	9%	2.1%

Table 3: Fields grouping input distribution per user

user names they have received. (Note that the stream in a topology using *fields grouping* is partitioned by the fields specified in the grouping.) The results can be seen in Table 3.

The evaluation test concludes as follows:

- If 10 or 20 users are allocated per *Bolt*, the uneven distribution is not relevant, since the volume of users is low and the server can handle that, even if all of the 20 users are very active at the same moment.
- For 100 or more users, the distribution of fields into the *Bolts* is approximately even and is optimal for the use cases for the system.

The evaluation results show that an implementation with *fields grouping* can be used to send all the events for one user to one *Bolt* consistently and process all of them in a designated *Bolt*. For the less than 100 users per *Bolt*, uneven distribution is not a concern because the low number of users produce a relatively low number of events that the system handles. For 100 users and more, it does not matter how active or inactive one specific user is, because the average behavior of all users will statistically even out the loads.

The scalability with respect to the large number of devices is not evaluated empirically with the *Socialite* system. However, the existing usage of the Apache Storm at Twitter with average 316 million active users and average 500 million tweets per day proves scalability. We expect that evenly distributing messages over many nodes in Apache Storm cluster will provide positive results on scalability in the *Socialite* architecture.

4.8.2.3 Extensibility Evaluation The extensibility of the architecture is evaluated with three different growth scenarios:

- *Adding a new device from a different manufacturer:* If the device type already exist in the semantic model but a new device from a different manufacturer needs to be added to the system, it is only necessary to provide a Java class with two functions: (a) one that extends the specific device model to provide the semantic information for such device, and (b) one that implements the corresponding device interface to handle the access to the device through the API provided by the manufacturer. Depending on the knowledge about the APIs relevant for the device manufacturer, the implementation time could be varied. During the *Socialite* proof of concept system development, we added NetAtMo temperature sensor after having Nest thermostat with temperature sensing capability. It took less than one week (40 person hours) for software developers who understand the *Socialite* architecture to add a new device from a new manufacturer.
- *Adding a new device type:* In order to add a new device type that does not exist in the current system, first a semantic model should be extended to support the new device type. Then the corresponding new device model class needs to be created by extending a generic device class together with the device capability interfaces to specify the functions (e.g., get temperature) modeled for the new device model. Once the device model class and device capability interfaces are defined, the rest of extension is same as the above extensibility scenario. The required time and effort for adding a new device type can be varied depending on the number of capabilities required for the new device type as well as the knowledge of the APIs from the manufacture. During the proof of concept, we added new device types over time because of the availability of the devices with us. It took less than two weeks (80 person hours) in average for software developers who understand the *Socialite* architecture to complete extending the new device type with testing.
- *Adding new rules by end users:* Although usability and user study are not in the scope of this dissertation, we evaluated the extensibility at the end user's end. To evaluate the extensibility, we registered five devices (including virtual devices), and asked the recruited end users (two with programming experience and two with no programming

experience) to create new rules after a short introduction of the system. The subjects came up with very creative rules (e.g., blinking the lamp with different colors to represent a Christmas light) as well as similar rules to what we had presented to them (e.g., if a motion is detected, then turn on the lamp and set the thermostat target temperature to their preferred temperature). Regardless of programming experience, all of them were able to complete the creation of their own rules within a 30 minutes evaluation session.

5.0 CONCLUSIONS AND FUTURE WORK

This dissertation proposed a novel scalable approach for smart home systems to connect, interact and share useful information through heterogeneous devices and people with common interests by realizing the new paradigm of the Social Internet of Things (SIoT).

To achieve an effective and novel collaboration between people and devices as well as among devices themselves, this dissertation created solutions to address the main challenges of SIoT, as follows. First, a system to provide interoperability of heterogeneous communication protocols in home networks and in application-level data models. Second, a mechanism for people and devices to effectively discover and share useful and relevant information at scale. Third, a collaboration framework for people and devices at many homes to use the discovered information toward an effective collaboration in an autonomous and customized way.

The presented work achieves the device interoperability in the home gateway and the cloud based collaboration framework by developing semantic models that abstract the heterogeneity of protocols and data models. The semantic models for users, locations, relationships, services and devices are a means to enable a uniform access to various connected devices and people in the system. An effective discovery and sharing of relevant information in SIoT is supported through newly defined social people-device and device-device relationships, which serve as a foundation for a new framework for the SIoT. Depending on the user's needs in different phases of the device life-cycle, different relationship types can be properly utilized such as when the new kinship relationship between devices with a same product model and manufacturer is used to discover and share repair history and solutions in the diagnosis/maintenance phase.

Furthermore, the reasoning framework introduced in this dissertation makes it easy for

end users to create their own rules and to share them with others having common interests. The distributed and scalable reasoning engines perform computationally intensive tasks to evaluate all rules upon a new event (e.g., device status change): distribution of reasoning engines over multiple nodes allows for parallel execution and scalability is achieved by using an open source data stream processing solution. The basic and low-level knowledge represented as semantic models is used when end users define their own high-level rules in order to make an automated decision in a cooperative manner.

In summary, this dissertation presented new theories and a proof of concept implementation through the use of real devices, accessible from protocol specifications and application programming interfaces to evaluate the proposed solution.

Beyond that, much work remains to be done. First, the promise of scalability of message handling needs to be better evaluated given the expected number of connected devices in IoT in the future. Second, the basic conflict resolution strategies need to be extended to encompass new and flexible mechanisms for multi-user and multi-device interactions. This can be accomplished without restricting user's creativity by predicting which newly-created rules cause conflicts. Third, achieving a common goal (e.g., energy saving) is implicitly possible in the presented solution by executing rules relevant to a common goal in the user's system. By adding new approaches such as gamification [151, 66], the system can explicitly visualize the performance and motivate users to voluntarily participate in a common goal. In addition, the management and the visualisation of end user created and sharable rules can be better supported to minimize the information overload when users have to deal with a number of different rules.

In the existing human based social networks research [152, 104, 39, 135, 74], social network analysis brought various benefits including the acceleration of knowledge flows, the improvement of efficiency and effectiveness of existing communication channels, and promoting peer supports. Although the SIoT is in an early phase of research, similar advantages are already identifiable when we grant new social relationships to the devices and people as discussed in new application types leveraging SIoT in this dissertation. The new collaboration framework for SIoT presented in this dissertation can bring similar benefits to the ones in human social network frameworks. Therefore, a new form of social capital obtained from a

new paradigm of SIoT should be further identified and researched.

Lastly, the user survey in this dissertation uncovers that the user's acceptance of the SIoT is highly related to how the system supports security and privacy, which is also supported by a survey from Pew Research Center [112]. Somewhat contradictorily, participants have demonstrated that they want to offer and get help with their devices and rules. The introduction of new relationships will open the discussion of balancing sharing/openness benefits and privacy risks from sharing, given the gap between their perceived privacy risks before and after joining the system. Note that privacy in SIoT will take different forms and will depend on device types (e.g., home appliances, health monitoring devices, security devices), different phases in the device life-cycle (e.g., operation, maintenance phases), data processing phases (communicating, processing, storing and sharing data), and interaction with devices and other users in different social relationships. Privacy enhancing solutions for the collaboration framework will be able to bootstrap a wide adoption of the new paradigm of the SIoT and ultimately help to gain a new form of social capital obtained from new social networks with people and connected devices.

BIBLIOGRAPHY

- [1] EnOcean Equipment Profiles (EEP) v2.0.
- [2] Facebook. Available: <https://www.facebook.com>.
- [3] Microformats. Available: <http://microformats.org/>.
- [4] OASIS device profile for web services (DPWS), 2009.
- [5] The Social Web of Things, 2012. Available: <https://www.youtube.com/watch?v=i5AuzQXBsG4>.
- [6] Social Networking Fact Sheet, 2014. Available: <http://www.pewinternet.org/fact-sheets/social-networking-fact-sheet/>.
- [7] Amazon Mechanical Turk, 2015. Available: <https://www.mturk.com/>.
- [8] Amazon Web Services, 2015. Available: <https://aws.amazon.com/>.
- [9] Atooma, 2015. Available: <http://www.atooma.com/>.
- [10] CES 2015 LG - smart home, 2015. Available: <https://www.youtube.com/watch?v=-AsuUdi1BiY>.
- [11] Google forms, 2015. Available: <https://www.google.com/forms>.
- [12] IFTTT, 2015. Available: <https://ifttt.com/>.
- [13] Jawbone APIs, 2015. Available: <https://jawbone.com/up/developer>.
- [14] NEST APIs, 2015. Available: <https://developer.nest.com>.
- [15] Philips Hue APIs, 2015. Available: <https://www.developers.meethue.com>.
- [16] Prosyst OSGi Services, 2015. Available: <http://www.prosyst.com>.
- [17] WigWag, 2015. Available: <http://wigwag.com/>.

- [18] Drools expert user guide, The JBoss Drools team. Available: <https://docs.jboss.org/drools/release/5.6.0.Final/drools-expert-docs/html/>.
- [19] Alessandro Acquisti and Ralph Gross. Imagined communities: Awareness, information sharing, and privacy on the Facebook. In *Proceedings of the 6th International Conference on Privacy Enhancing Technologies*, PET'06, pages 36–58, Berlin, Heidelberg, 2006. Springer-Verlag.
- [20] Gail-Joon Ahn, Hongxin Hu, and Jing Jin. Security-enhanced OSGi service environments. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 39(5):562–571, Sept 2009.
- [21] Dean Allemang and James Hendler. *Semantic Web for the working ontologist: effective modeling in RDFS and OWL*. Elsevier, 2011.
- [22] OSGi Alliance. OSGi service platform release 4 version 4.2 core specification, 2009.
- [23] Z-Wave Alliance. Z-wave specification. 2015.
- [24] ZigBee Alliance. Zigbee home automation public application profile. *IEEE J. Select. Areas Commun*, 2007.
- [25] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM, 2011.
- [26] Konnex Association et al. Knx specification, 2004.
- [27] Luigi Atzori, Davide Carboni, and Antonio Iera. Smart things in the social loop: Paradigms, technologies, and potentials. *Ad Hoc Networks*, 18:121–132, 2014.
- [28] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010.
- [29] Luigi Atzori, Antonio Iera, and Giacomo Morabito. SIoT: Giving a social structure to the internet of things. *Communications Letters, IEEE*, 15(11):1193–1195, 2011.
- [30] Luigi Atzori, Antonio Iera, Giacomo Morabito, and Michele Nitti. The social internet of things (SIOT)–When social networks meet the internet of things: Concept, architecture and network characterization. *Computer Networks*, 56(16):3594–3608, 2012.
- [31] Franz Baader and Ulrike Sattler. An overview of tableau algorithms for description logics. *Studia Logica*, 69(1):5–40, 2001.
- [32] Matthew Ball and Vic Callaghan. Managing control, convenience and autonomy-a study of agent autonomy in intelligent environments., 2012.

- [33] M Baqer. Enabling collaboration and coordination of wireless sensor networks via social networks. In *Distributed Computing in Sensor Systems Workshops (DCOSSW), 2010 6th IEEE International Conference on*, pages 1–2. IEEE, 2010.
- [34] Sean Bechhofer. Owl: Web ontology language. In *Encyclopedia of Database Systems*, pages 2008–2009. Springer, 2009.
- [35] Dave Beckett and Brian McBride. RDF/XML syntax specification (revised). *W3C recommendation*, 10, 2004.
- [36] David Beckett. N-Triples: A line-based syntax for an RDF graph, 2013.
- [37] David Beckett, Tim Berners-Lee, and Eric Prud’hommeaux. Turtle-terse RDF triple language. *W3C Team Submission*, 14, 2008.
- [38] Vicenc Beltran, Antonio M Ortiz, Dina Hussein, and Noel Crespi. A semantic service creation platform for social iot. In *Internet of Things (WF-IoT), 2014 IEEE World Forum on*, pages 283–286. IEEE, 2014.
- [39] Jacqueline L Bender, Maria-Carolina Jimenez-Marroquin, and Alejandro R Jadad. Seeking support on facebook: a content analysis of breast cancer groups. *Journal of medical Internet research*, 13(1), 2011.
- [40] Tim Berners-Lee and Dan Connolly. Notation3 (n3): A readable RDF syntax. W3C Team Submission (Mar 2011).
- [41] Tim Berners-Lee, James Hendler, Ora Lassila, et al. The semantic web. *Scientific american*, 284(5):28–37, 2001.
- [42] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data-the story so far. *Semantic Services, Interoperability and Web Applications: Emerging Concepts*, pages 205–227, 2009.
- [43] Bluetooth SIG, Inc. Bluetooth smart home, 2004.
- [44] Jürgen Bock, Peter Haase, Qiu Ji, and Raphael Volz. Benchmarking OWL reasoners. In *Proc. of the ARea2008 Workshop, Tenerife, Spain (June 2008)*, 2008.
- [45] Dario Bonino and Fulvio Corno. *Dogont-ontology modeling for intelligent domotic environments*. Springer, 2008.
- [46] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16. ACM, 2012.
- [47] Mike Botts, George Percivall, Carl Reed, and John Davidson. OGC sensor web enablement: Overview and high level architecture. In *GeoSensor networks*, pages 175–190. Springer, 2008.

- [48] Mike Botts and Alexandre Robin. OpenGIS sensor model language (SensorML) implementation specification. *OpenGIS Implementation Specification OGC*, 7(000), 2007.
- [49] Ronald Brachman and Hector Levesque. *Knowledge representation and reasoning*. Elsevier, 2004.
- [50] Dan Brickley and Ramanathan V Guha. RDF vocabulary description language 1.0: Rdf schema. 2004.
- [51] Dan Brickley and Libby Miller. Foaf vocabulary specification 0.98. *Namespace document*, 9, 2012.
- [52] Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *The Semantic WebISWC 2002*, pages 54–68. Springer, 2002.
- [53] AJ Brush, Bongshin Lee, Ratul Mahajan, Sharad Agarwal, Stefan Saroiu, and Colin Dixon. Home automation in the wild: challenges and opportunities. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2115–2124. ACM, 2011.
- [54] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.
- [55] Harry Chen, Filip Perich, Tim Finin, and Anupam Joshi. Soupa: Standard ontology for ubiquitous and pervasive applications. In *Mobile and Ubiquitous Systems: Networking and Services, 2004. MOBIQUITOUS 2004. The First Annual International Conference on*, pages 258–267. IEEE, 2004.
- [56] Marshini Chetty, David Tran, and Rebecca E Grinter. Getting to green: understanding resource consumption in the home. In *Proceedings of the 10th international conference on Ubiquitous computing*, pages 242–251. ACM, 2008.
- [57] Eun Cho, Chang-Joo Moon, and Doo-Kwon Baik. Home gateway operating model using reference monitor for enhanced user comfort and privacy. *Consumer Electronics, IEEE Transactions on*, 54(2):494–500, 2008.
- [58] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana, et al. Web services description language (WSDL) 1.1, 2001.
- [59] Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl GarcíA-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, et al. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012.
- [60] Diane J. Cook, Michael Youngblood, Edwin O. Heierman, III, Karthik Gopalratnam, Sira Rao, Andrey Litvin, and Farhan Khawaja. Mavhome: An agent-based smart home. In *Proceedings of the First IEEE International Conference on Pervasive Computing*

and Communications, PERCOM '03, pages 521–, Washington, DC, USA, 2003. IEEE Computer Society.

- [61] Taylor G Cowan. Jenabean: Easily bind JavaBeans to RDF. *IBM DeveloperWorks*, 2008.
- [62] Paul Darbee. Insteon: The details. *Smarthome Technology*, pages 1–64, 2005.
- [63] Scott Davidoff, Min Kyung Lee, Charles Yiu, John Zimmerman, and Anind K Dey. Principles of smart home control. In *UbiComp 2006: Ubiquitous Computing*, pages 19–34. Springer, 2006.
- [64] Scott Davidoff, Min Kyung Lee, John Zimmerman, and AK Dey. Socially-aware requirements for a smart home. In *Proceedings of the international symposium on intelligent environments*, pages 41–44. Citeseer, 2006.
- [65] Luigi De Russis and Fulvio Corno. Homerules: A tangible end-user programming interface for smart homes. In *Proceedings of the 33rd Annual ACM Conference Extended Abstracts on Human Factors in Computing Systems*, pages 2109–2114. ACM, 2015.
- [66] Sebastian Deterding. Gamification: designing for motivation. *interactions*, 19(4):14–17, 2012.
- [67] Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.
- [68] Anind K Dey, Timothy Sohn, Sara Streng, and Justin Kodama. icap: Interactive prototyping of context-aware applications. In *Pervasive Computing*, pages 254–271. Springer, 2006.
- [69] Andreas Dieberger. Supporting social navigation on the World Wide Web. *International Journal of Human-Computer Studies*, 46(6):805–825, 1997.
- [70] Pavlin Dobrev, David Famolari, Christian Kurzke, Brent Miller, et al. Device and service discovery in home networks with osgi. *Communications Magazine, IEEE*, 40(8):86–92, 2002.
- [71] Paul Dourish and Matthew Chalmers. Running out of space: Models of information navigation. In *Short paper presented at HCI*, volume 94, pages 23–26, 1994.
- [72] Bob DuCharme. *Learning Sparql*. O'Reilly Media, Inc., 2013.
- [73] Markus Eisenhauer, Peter Rosengren, and Pablo Antolin. Hydra: A development platform for integrating wireless devices and sensors into ambient intelligence systems. In *The Internet of Things*, pages 367–373. Springer, 2010.

- [74] Nicole B Ellison, Charles Steinfield, and Cliff Lampe. Connection strategies: Social capital implications of Facebook-enabled communication practices. *New media & society*, page 1461444810385389, 2011.
- [75] Orri Erling and Ivan Mikhailov. RDF support in the Virtuoso DBMS. In *Networked Knowledge-Networked Media*, pages 7–24. Springer, 2009.
- [76] Foundation for Intelligent Physical Agents. FIPA Device Ontology Specification. Available: <http://www.fipa.org/specs/fipa00091/>.
- [77] Charles L Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial intelligence*, 19(1):17–37, 1982.
- [78] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.
- [79] N Georgantas et al. Amigo middleware core: Prototype implementation & documentation. *IST Amigo Project Deliverable D*, 3:2, 2006.
- [80] W3C OWL Working Group et al. OWL 2 Web Ontology Language document overview. 2009.
- [81] Thomas R Gruber. A translation approach to portable ontology specifications. *Knowledge acquisition*, 5(2):199–220, 1993.
- [82] Tao Gu, Hung Keng Pung, and Da Qing Zhang. Toward an osgi-based infrastructure for context-aware applications. *Pervasive Computing, IEEE*, 3(4):66–74, 2004.
- [83] Tao Gu, Hung Keng Pung, and Da Qing Zhang. A service-oriented middleware for building context-aware services. *Journal of Network and computer applications*, 28(1):1–18, 2005.
- [84] Tao Gu, Hung Keng Pung, Da Qing Zhang, Hung Keng Pung, and Da Qing Zhang. *A bayesian approach for dealing with uncertain contexts*. na, 2004.
- [85] Dominique Guinard, Mathias Fischer, and Vlad Trifa. Sharing using social networks in a composable web of things. In *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, pages 702–707. IEEE, 2010.
- [86] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences), Madrid, Spain*, page 15, 2009.
- [87] Young-Guk Ha. Dynamic integration of zigbee home networks into home gateways using osgi service registry. *Consumer Electronics, IEEE Transactions on*, 55(2):470–476, 2009.

- [88] Jing Han, E Haihong, Guan Le, and Jian Du. Survey on nosql database. In *Pervasive computing and applications (ICPCA), 2011 6th international conference on*, pages 363–366. IEEE, 2011.
- [89] Dick Hardt. The oauth 2.0 authorization framework. 2012.
- [90] Markus C Huebscher and Julie A McCann. Adaptive middleware for context-aware applications in smart-homes. In *Proceedings of the 2nd workshop on Middleware for pervasive and ad-hoc computing*, pages 111–116. ACM, 2004.
- [91] Apache Jena. Semantic Web framework for Java, 2007.
- [92] Apache Jena. Apache jena. *jena. apache. org [Online]*. Available: <http://jena.apache.org> [Accessed: Mar. 20, 2014], 2013.
- [93] Aman Kansal, Suman Nath, Jie Liu, and Feng Zhao. Senseweb: An infrastructure for shared sensing. *IEEE multimedia*, (4):8–13, 2007.
- [94] Fahim Kawsar, Tatsuo Nakajima, and Kaori Fujinami. Deploy spontaneously: supporting end-users in building and enhancing a smart home. In *Proceedings of the 10th international conference on Ubiquitous computing*, pages 282–291. ACM, 2008.
- [95] Jan H Kietzmann, Kristopher Hermkens, Ian P McCarthy, and Bruno S Silvestre. Social media? get serious! understanding the functional building blocks of social media. *Business horizons*, 54(3):241–251, 2011.
- [96] Ji Eun Kim. Integrify System Demo, 2015. Available: <https://youtu.be/eE5QLZJELzU>.
- [97] Ji Eun Kim. Socialite Application Demo, 2015. Available: <http://youtu.be/B1C01UXVCqY>.
- [98] Ji Eun Kim, Tassilo Barth, George Boulos, John Yackovich, Christian Beckel, and Daniel Mosse. Seamless integration of heterogeneous devices and access control in smart homes and its evaluation. *Intelligent Buildings International*, (ahead-of-print):1–18, 2015.
- [99] Ji Eun Kim, George Boulos, John Yackovich, Tassilo Barth, Christian Beckel, and Daniel Mosse. Seamless integration of heterogeneous devices and access control in smart homes. In *Intelligent Environments (IE), 2012 8th International Conference on*, pages 206–213. IEEE, 2012.
- [100] Ji Eun Kim, Adriano Maron, and Daniel Mosse. Socialite: A flexible framework for social internet of things. In *Mobile Data Management (MDM), 2015 16th IEEE International Conference on*, volume 1, pages 94–103. IEEE, 2015.
- [101] Tiffany Hyun-Jin Kim, Lujo Bauer, James Newsome, Adrian Perrig, and Jesse Walker. Challenges in access right assignment for secure home networks. In *HotSec*, 2010.

- [102] Atanas Kiryakov, Damyan Ognyanov, and Dimitar Manov. Owlīm—a pragmatic semantic repository for owl. In *Web Information Systems Engineering–WISE 2005 Workshops*, pages 182–192. Springer, 2005.
- [103] Graham Klyne and Jeremy J Carroll. Resource description framework (RDF): Concepts and abstract syntax. 2006.
- [104] David Knoke and Song Yang. *Social network analysis*, volume 154. Sage, 2008.
- [105] Jonathon Kopecky, Karthik Gomadam, and Tomas Vitvar. hrests: An html microformat for describing restful web services. In *Web Intelligence and Intelligent Agent Technology, 2008. WI-IAT’08. IEEE/WIC/ACM International Conference on*, volume 1, pages 619–625. IEEE, 2008.
- [106] Jonathon Kopecky, Tomas Vitvar, Carine Bournez, and Joel Farrell. SAWSDL: Semantic annotations for WSDL and XML schema. *Internet Computing, IEEE*, 11(6):60–67, 2007.
- [107] Choonhwa Lee, David Nordstedt, and Sumi Helal. Enabling smart spaces with osgi. *Pervasive Computing, IEEE*, 2(3):89–94, 2003.
- [108] Jonathan Leibiusky, Gabriel Eisbruch, and Dario Simonassi. *Getting started with storm*. O’Reilly Media, Inc., 2012.
- [109] Dong Liu, Carlos Pedrinaci, and John Domingue. A framework for feeding linked data to complex event processing engines. 2010.
- [110] David Luckham. *The power of events*, volume 204. Addison-Wesley Reading, 2002.
- [111] David C Luckham. *Event processing for business: organizing the real-time enterprise*. John Wiley & Sons, 2011.
- [112] Mary Madden. Public perceptions of privacy and security in the post-snowden era, 2014. <http://www.pewinternet.org/2014/11/12/public-privacy-perceptions/>.
- [113] Cliff Lampe Amanda Lenhart Mary Madden Maeve Duggan, Nicole B. Ellison. Social media update 2014. Available: <http://www.pewinternet.org/2015/01/09/social-media-update-2014/>.
- [114] Niko Mäkitalo, Jari Pääkkö, Mikko Raatikainen, Varvana Myllärniemi, Timo Aaltonen, Tapani Leppänen, Tomi Männistö, and Tommi Mikkonen. Social devices: collaborative co-located interactions in a mobile cloud. In *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, page 10. ACM, 2012.
- [115] Jennifer Mankoff, Deanna Matthews, Susan R Fussell, and Michael Johnson. Leveraging social networks to motivate individuals to reduce their ecological footprints. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 87–87. IEEE, 2007.

- [116] Dave Marples and Peter Kriens. The open services gateway initiative: An introductory overview. *Communications Magazine, IEEE*, 39(12):110–114, 2001.
- [117] David Martin, Mark Burstein, Jerry Hobbs, Ora Lassila, Drew McDermott, Sheila McIlraith, Sridhar Narayanan, Massimo Paolucci, Bijan Parsia, Terry Payne, et al. OWL-S: Semantic markup for web services. *W3C member submission*, 22:2007–04, 2004.
- [118] Jeff McAffer, Paul VanderLei, and Simon Archer. *OSGi and Equinox: Creating highly modular Java systems*. Addison-Wesley Professional, 2010.
- [119] Deborah L McGuinness, Frank Van Harmelen, et al. OWL Web ontology language overview. *W3C recommendation*, 10(10), 2004.
- [120] Peter Membrey, Eelco Plugge, and DUPTim Hawkins. *The definitive guide to MongoDB: the NoSQL database for cloud and desktop computing*. Apress, 2010.
- [121] Brenda M Michelson. Event-driven architecture overview. *Patricia Seybold Group*, 2, 2006.
- [122] Peter Middleton, Peter Kjeldsen, and Jim Tully. Forecast: The internet of things, worldwide, 2013. *Gartner Research*, 2013.
- [123] Daniele Miorandi, Sabrina Sicari, Francesco De Pellegrini, and Imrich Chlamtac. Internet of things: Vision, applications and research challenges. *Ad Hoc Networks*, 10(7):1497–1516, 2012.
- [124] Tim Moses et al. Extensible access control markup language (xacml) version 2.0. *Oasis Standard*, 200502, 2005.
- [125] Catherine Moxey, Mike Edwards, Opher Etzion, Mamdouh Ibrahim, Sreekanth Iyer, Hubert Lalanne, Mweene Monze, Marc Peters, Yuri Rabinovich, Guy Sharon, et al. A conceptual model for event processing systems. *IBM Redguide publication*, 2010.
- [126] Alan J Munro, Kristina Höök, and David Benyon. *Social navigation of information space*. Springer Science & Business Media, 2012.
- [127] Netatmo. NetAtMo APIs, 2015. Available: <https://dev.netatmo.com>.
- [128] Mark W Newman. Now we’re cooking: Recipes for end-user service composition in the digital home. 2006.
- [129] Mark W Newman, Ame Elliott, and Trevor F Smith. Providing an integrated user experience of networked media, devices, and services through end-user composition. In *Pervasive Computing*, pages 213–227. Springer, 2008.
- [130] Huansheng Ning and Ziou Wang. Future internet of things architecture: like mankind neural system or social organization framework? *Communications Letters, IEEE*, 15(4):461–463, 2011.

- [131] Natalya F Noy. Semantic integration: a survey of ontology-based approaches. *ACM Sigmod Record*, 33(4):65–70, 2004.
- [132] GIS Open. Observation and Measurements (O&M) Implementation Specification. *OGC Geospatial Consortium INC*, 2007.
- [133] Antonio M Ortiz, Dina Hussein, Soochang Park, Son N Han, and Noel Crespi. The cluster between internet of things and social networks: Review and research challenges. *Internet of Things Journal, IEEE*, 1(3):206–215, 2014.
- [134] Addy Osmani. *Developing Backbone.js Applications*. O’Reilly Media, Inc., 2013.
- [135] Namsu Park, Kerk F Kee, and Sebastián Valenzuela. Being immersed in social networking environment: Facebook groups, uses and gratifications, and social outcomes. *CyberPsychology & Behavior*, 12(6):729–733, 2009.
- [136] Vctor Pelez, Roberto Gonzlez, Luis ngel San Martn, Antonio Campos, and Vanesa Lobato. *Multilevel and hybrid architecture for device abstraction and context information management in smart home environments*, pages 207–216. Springer, 2010.
- [137] Antonio Pintus, Davide Carboni, and Andrea Piras. Paraimpu: a platform for a social web of things. In *Proceedings of the 21st international conference companion on World Wide Web*, pages 401–404. ACM, 2012.
- [138] Alan Presser, Lee Farrell, Devon Kemp, and W Lupton. Upnp device architecture 1.1. In *UPnP Forum*, volume 22, 2008.
- [139] Seth Proctor et al. Sun’s xacml implementation. *sunxacml. sourceforge. net*, 2004.
- [140] Eric Prud’Hommeaux, Andy Seaborne, et al. SPARQL query language for RDF. *W3C recommendation*, 15, 2008.
- [141] Parisa Rashidi and Diane J Cook. Keeping the resident in the loop: Adapting the smart home to the user. *Systems, Man and Cybernetics, Part A: Systems and Humans, IEEE Transactions on*, 39(5):949–959, 2009.
- [142] Christian Reinisch, Mario J Kofler, Félix Iglesias, and Wolfgang Kastner. Thinkhome energy efficiency in future smart homes. *EURASIP Journal on Embedded Systems*, 2011:1, 2011.
- [143] Mark Richards, Richard Monson-Haefel, and David A Chappell. *Java message service*. O’Reilly Media, Inc., 2009.
- [144] Leonard Richardson and Sam Ruby. *RESTful web services*. O’Reilly Media, Inc., 2008.
- [145] Alejandro Rodriguez, Robert McGrath, Yong Liu, James Myers, and I Urbana-Champaign. Semantic management of streaming data. *Proc. Semantic Sensor Networks*, 80, 2009.

- [146] Dumitru Roman, Uwe Keller, Holger Lausen, Jos de Bruijn, Rubén Lara, Michael Stollberg, Axel Polleres, Cristina Feier, Christoph Bussler, Dieter Fensel, et al. Web service modeling ontology. *Applied ontology*, 1(1):77–106, 2005.
- [147] Dave Rye. The x10 powerhouse powerline interface. Technical report, Technical report, X10 PowerHouse, 2001.
- [148] Ravi Sandhu and Qamar Munawer. How to do discretionary access control using roles. In *Proceedings of the third ACM workshop on Role-based access control*, pages 47–54. ACM, 1998.
- [149] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-based access control models. *Computer*, (2):38–47, 1996.
- [150] Martin Sarnovský, Peter Kostelník, Ján Hreňo, and Peter Butka. Device description in hydra middleware. In *Proceedings of the 2nd Workshop on Intelligent and Knowledge oriented Technologies*, pages 71–74, 2007.
- [151] Jesse Schell. *The Art of Game Design: A book of lenses*. CRC Press, 2014.
- [152] John Scott. *Social network analysis*. Sage, 2012.
- [153] Amit Sheth, Cory Henson, and Satya S Sahoo. Semantic sensor web. *Internet Computing, IEEE*, 12(4):78–83, 2008.
- [154] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical owl-dl reasoner. *Web Semantics: science, services and agents on the World Wide Web*, 5(2):51–53, 2007.
- [155] Bruce Snyder, Dejan Bosnanac, and Rob Davies. *ActiveMQ in action*. Manning, 2011.
- [156] Lorenzo Sommaruga, Antonio Perri, and Francesco Furfari. DomoML-env: an ontology for Human Home Interaction. In *SWAP*, volume 166. Citeseer, 2005.
- [157] S Sorrell. Smart home ecosystems & the internet of things-strategies & forecasts 2014-2018. *Juniperresearch. com*. Retrieved August, 29:2014, 2014.
- [158] Manu Sporny, Dave Longley, Gregg Kellogg, Markus Lanthaler, and Niklas Lindström. JSON-LD 1.0. *W3C Recommendation (January 16, 2014)*, 2014.
- [159] Apache Storm. Storm, Distributed and Fault-Tolerant Real-time Computation. 2014.
- [160] ITU Strategy and Policy Unit. Itu internet reports 2005: The internet of things. *Geneva: International Telecommunication Union (ITU)*, 2005.
- [161] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al.

- Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.
- [162] RDF Turtle-Terse. Triple language. *W3C Team Submission*, 14, 2008.
 - [163] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. Practical trigger-action programming in the smart home. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 803–812. ACM, 2014.
 - [164] Dimitar Valtchev and Ivailo Frankov. Service gateway architecture for a smart home. *Communications Magazine, IEEE*, 40(4):126–132, 2002.
 - [165] Tam Van Nguyen, Wontaek Lim, Huy Nguyen, Deokjai Choi, and Chilwoo Lee. Context ontology implementation for smart home. *arXiv preprint arXiv:1007.1273*, 2010.
 - [166] Xiao Hang Wang, Da Qing Zhang, Tao Gu, and Hung Keng Pung. Ontology based context modeling and reasoning using owl. In *Pervasive Computing and Communications Workshops, 2004. Proceedings of the Second IEEE Annual Conference on*, pages 18–22. Ieee, 2004.
 - [167] Zhiqiang Wei, Shuwei Qin, Dongning Jia, and Yongquan Yang. Research and design of cloud architecture for smart home. In *Software Engineering and Service Sciences (ICSESS), 2010 IEEE International Conference on*, pages 86–89. IEEE, 2010.
 - [168] Chao-Lin Wu, Chun-Feng Liao, and Li-Chen Fu. Service-oriented smart-home architecture based on osgi and mobile-agent technology. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 37(2):193–205, 2007.
 - [169] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 407–418. ACM, 2006.
 - [170] Tao Zhang and Bernd Brügge. Empowering the user to build smart home applications. In *ICOST 2004 International Conference on Smart Home and Health Telematics*, 2004.